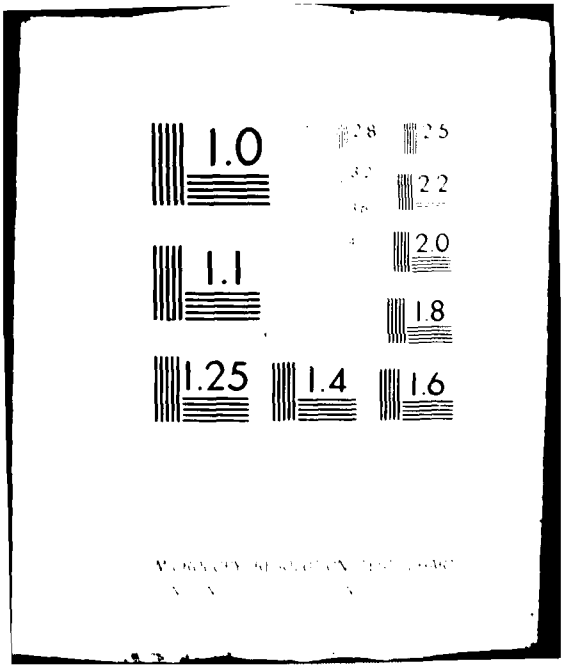
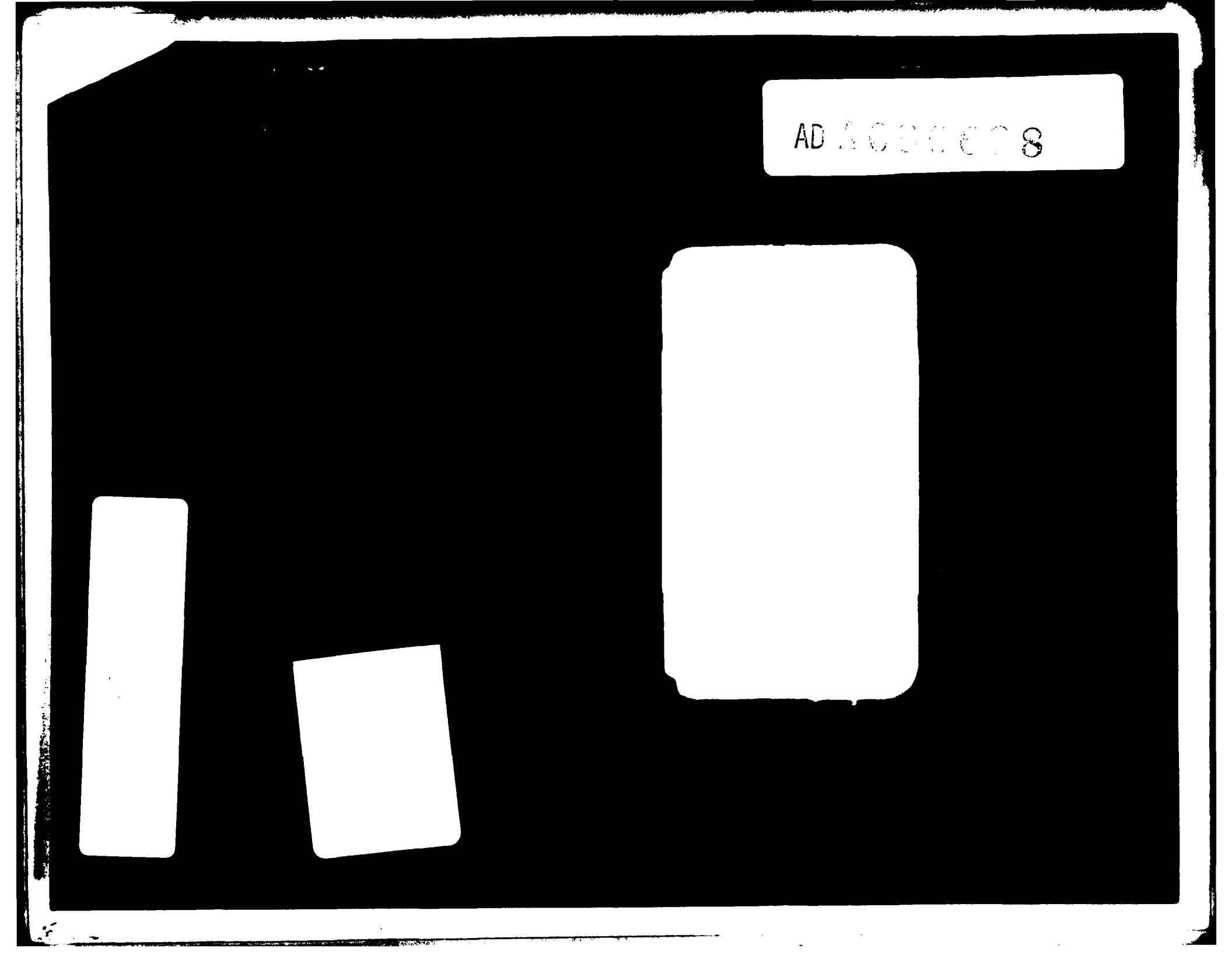
AD-A090 688 WASHINGTON UNIV SEATTLE DEPT OF COMPUTER SCIENCE F/G 12/1
DECIDABILITY AND EXPRESSIVENESS OF LOGICS OF PROCESSES.(U)
AUG 80 K R ABRAHAMSON N00014-80-C-0221
UNCLASSIFIED TR-80-08-01 NL

1 OF 2
AD A 090688


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5


AD

Decidability and Expressiveness of Logics of Processes*

by

Karl Raymond Abrahamson
(Ph.D. Thesis)

Technical Report #80-08-01

*This research was supported in part by Office of Naval Research Contract N00014-80-C-0221 with the University of Washington.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: R-80-08-01	2. GOVT ACCESSION NO.: AD-A090688
3. RECIPIENT'S CATALOG NUMBER: 12 178	
4. TITLE (and Subtitle): DECIDABILITY AND EXPRESSIVENESS OF LOGICS OF PROCESSES, (Ph.D. Thesis)	5. TYPE OF REPORT & PERIOD COVERED: Technical Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Karl Raymond Abrahamson	8. CONTRACT OR GRANT NUMBER(s): ONR Contract: N00014-80-C-0221 new
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Department of Computer Science, FR-35 University of Washington Seattle, WA 98195	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: NR 049-456/30 Oct 79 (437)
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research 800 N. Quincy Arlington, VA 22217 Att: Dr. R. B. Grafton	12. REPORT DATE: Aug 1980 13. NUMBER OF PAGES: 177
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): 9 Doctoral thesis	15. SECURITY CLASS. (of this report): Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Process logic, dynamic logic, temporal logic, decidability, completeness, expressiveness.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We define and study several logics of processes. The logics GPL and MPL are based on a second order tense logic, where the two types of variable range over computation sequences and points on computation sequences. GPL is a version of the predicate calculus, similar to Parikh's general logic. MPL is a modal logic, and is the only modal process logic we know of which incorporates two fundamentally different types of modality. When syntactic programs are included in MPL, MPL is at least as expressive as PDL+, Parikh's

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102 LF 014 6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AC 315224

SOAPL, Pnueli's tense logic or Nishimura's process logic, and contains both Lamport's linear and branching time logics.

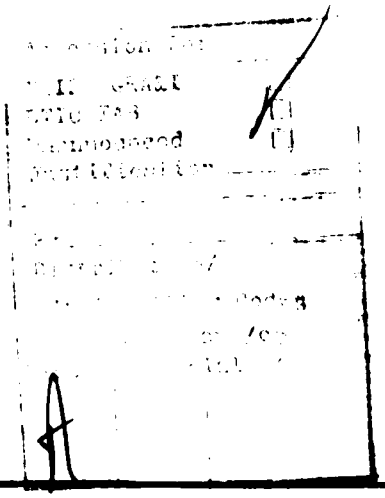
We present a tableau method for deciding validity in MPL, based on a new type of directed graph, called an LL-graph. From the tableau method we derive a complete proof system for MPL.

Although GPL and MPL are based on the same notions, we find some interesting differences between the two. MPL is decidable in double exponential time, while even a proper subset of GPL, which can express the same properties as MPL, is nonelementary. We are able to show that GPL is decidable only when processes are tree-like, in Parikh's sense. In contrast, our method for deciding MPL in general requires processes which are not tree-like.

Processes are defined on a very abstract level, as sets of computation sequences. Intrinsic to our definition of a process is the notion of deadlock. Both GPL and MPL have provisions for explicitly discussing deadlock, which most other process logics to date ignore.

We also study extensions to PDL. We show, provided only that basic programs are indivisible actions, that extending PDL by a concurrency operator, a global invariance operator and flowgraph programs, among others, adds no expressive power to PDL. Moreover, there is a better way to decide formulas in the extended logic than to translate them to PDL. We extend PDL by adding special Boolean variables, which can be set and tested. Boolean variable PDL efficiently simulates the above extensions, and is shown to be decidable by a faster method than by eliminating Boolean variables.

We prove a lower bound on the complexity of B-PDL which is a function of two parameters, the length of the input, and the number of variables it contains. The proof involves a compression theorem for functions of several variables, which may be of general use.



University of Washington

Abstract

DECIDABILITY AND EXPRESSIVENESS
OF LOGIC AND PROCESSES

By Karl Raymond Abrahamson

Chairperson of the Supervisory Committee:
Professor Michael J. Fischer
Department of Computer Science

We define and study several logics of processes. The logics GPL and MPL are based on a second order tense logic, where the two types of variable range over computation sequences and points on computation sequences. GPL is a version of the predicate calculus, similar to Parikh's general logic. MPL is a modal logic, and is the only modal process logic we know of which incorporates two fundamentally different types of modality. When syntactic programs are included in MPL, MPL is at least as expressive as PDL$^+$, Parikh's SOAPL, Pnueli's tense logic or Nishimura's process logic, and contains both Lamport's linear and branching time logics.

We present a tableau method for deciding validity in MPL, based on a new type of directed graph, called an LL-graph. From the tableau method we derive a complete proof system for MPL.

Although GPL and MPL are based on the same notions, we find some interesting differences between the two. MPL is decidable in double exponential time, while even a proper subset of GPL, which can express the same properties

as MPL, is nonelementary. We are able to show that GPL is decidable only when processes are tree-like, in Parikh's sense. In contrast, our method for deciding MPL in general requires processes which are not tree-like.

Processes are defined on a very abstract level, as sets of computation sequences. Intrinsic to our definition of a process is the notion of deadlock. Both GPL and MPL have provisions for explicitly discussing deadlock, which most other process logics to date ignore.

We also study extensions to PDL. We show, provided only that basic programs are indivisible actions, that extending PDL by a concurrency operator, a global invariance operator and flowgraph programs, among others, adds no expressive power to PDL. Moreover, there is a better way to decide formulas in the extended logic than to translate them to PDL. We extend PDL by adding special Boolean variables, which can be set and tested. Boolean variable PDL efficiently simulates the above extensions, and is shown to be decidable by a faster method than by eliminating Boolean variables.

We prove a lower bound on the complexity of B-PDL which is a function of two parameters, the length of the input, and the number of variables it contains. The proof involves a compression theorem for functions of several variables, which may be of general use.

I would like to express my thanks to Professor Fischer for his patience and many hours spent discussing this work.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

Chapter 1

Introduction

In recent years a great deal of effort has gone into discovering convenient and powerful methods of reasoning about the behavior of computer programs. There are two main goals of this research. First, we need a precise definition of exactly what a program is. At present there is no general agreement on the exact meanings of programs, and there is even less agreement on what sort of programs we should be assigning meaning to. Second, we need a convenient but precise method of proving properties of programs. Even when the meaning of a program is understood, the very general set-theoretic proofs have proved cumbersome, with most authors choosing more informal methods. The results have been incorrect or unconvincing proofs. For example, Dijkstra's on-the-fly garbage collector [D78] in its original version contained a subtle bug, although Dijkstra "proved" the program correct.

Below is a brief history of the work leading up to this work.

Floyd [F167] and Hoare [Ho69] presented early systems for reasoning about programs. Those methods are used primarily for proving properties related to termination of the program. For example, Hoare's partial correctness

assertion P {A} Q states that if program A is started with P true, then whenever (if ever) A terminates, Q holds. Floyd suggests the well-founded-set method of proving that a program must terminate, which consists of showing that going around any loops in the program must result in the decrease of some well founded quantity.

Partial correctness is far from the only useful property of programs. Manna and Waldinger [MW78] give examples where using the condition "P must eventually become true" leads to natural proofs of interesting properties of programs. A really useful logic of programs should permit its user many different methods of reasoning about programs. Pratt's Dynamic Logic [Pr76] and later Harel's DL^+ [HP78] bring the "eventuality" and partial correctness methods together into a single elegant framework. The heart of DL is the formula [A]Q, meaning "if program A is started in the current state, then whenever (if ever) A terminates, Q holds." The Hoare style partial correctness assertion P {A} Q is expressed in DL as $P \supset [A]Q$, which simply states that, if P holds in the current state, then [A]Q also holds in the current state. The dual $\langle A\rangle Q \equiv \sim[A]\sim Q$ of [A]Q states that it is possible for program A to halt with Q true. Dynamic Logic programs are in general nondeterministic. Hence it is possible for $\langle A\rangle Q$ and $\langle A\rangle \sim Q$ to be simultaneously true.

Among concurrent programs, programs which terminate

are the exception rather than the rule. Typical nonterminating programs are operating systems, on-the-fly garbage collectors, the dining philosophers program, and so on (see [FP76]). It is clear that termination properties are inadequate for reasoning about such programs. Pratt [Pr78] suggests extending Dynamic Logic by adding new operators for discussing the behavior of a program in time. For instance, the operator {A}Q expresses the global invariance of Q over A, meaning that Q holds throughout the execution of program A, started in the current state. Numerous other properties are possible.

Among possible operators for describing the temporal behavior of programs, Lamport [L80] identifies two classes: linear time and branching time operators. Most logics to date include either one or the other, but not both. As both have uses, a powerful logic should include both.

By process logic, we mean any language which is used to express properties of processes, or programs, the properties in general not being related to the termination of the process. We have mentioned the process logics of Pratt and Lamport. Others, which are described in more detail later, are the process logics of Pnueli [Pn77, Pn79], Gabbay et al., [GPSS80], Parikh [Pa78], Harel et al., [HKP80], and Nishimura [N79].

In this work we take three approaches to process logic.

1. What sorts of properties can be expressed in a simple, termination oriented logic, in particular Propositional Dynamic Logic (PDL)? In Chapter 2 we demonstrate that PDL can express much more than is readily apparent. The power of PDL is revealed by adding auxiliary Boolean variables to PDL. Such variables add no expressive power to PDL, though they allow more concise expression of some properties. In particular, properties regarding the concurrent execution of programs can be expressed concisely using Boolean variables.

While PDL can express a surprising number of properties of programs, it cannot express all that we need. Therefore we develop more powerful logics.

2. The second approach is the classical approach of defining a version of the predicate calculus which is suited to describing processes. We call this logic GPL, for General Process Logic. Unlike PDL, GPL does not have programs -- a valid GPL sentence is one which holds for all processes. The absence of programs makes the presentation of GPL simpler, and allows us to at least partially analyze GPL.

3. The third approach is to adapt modal logic to a logic of processes. The logic MPL (for Modal Process Logic) is slightly less expressive than GPL, but is much easier to analyze, and to work with in general. We prove

that MPL is decidable, and give a complete proof system for MPL.

Neither GPL nor MPL has programs. In Chapter 5 we consider the addition of programs to MPL. (Programs can also be added to GPL, but we do not bother to define GPL with programs here.) MPL with programs is called MPL/P. MPL/P has at least as much expressive power as Nishimura's process logic [N79], which in turn is at least as expressive at Pratt's process logic [Pr78], and Parikh's SOAPL [Pa78]. We conjecture that MPL/P is more expressive than all of the above logics.

1.1. Processes

The rest of this chapter is spent defining processes and programs and discussing the consequences of those definitions. Of primary importance is the discussion of blocking, which may differ from the reader's notion.

A process is a semantic entity, as opposed to a program, which is syntactic. We choose a very abstract notion of process. There are no communication primitives, as there are in [Ho76, MM77]. Instead, individual processes communicate with each other by altering a common state, which can be thought of as encompassing all of the memory of the system, whether private to a given process or shared by two or more processes. Indeed, there is no

notion of several processes inherent to the semantics of processes. The definition of a process is sufficiently general that an entire system of processes running concurrently can be viewed as a single super process.

Our notion of process is related to Pratt's, in that a process is a set of computation sequences over a given set of states U. The main difference is that rather than being a sequence of states, a computation sequence, or path, is a sequence of transitions between states. The transition from state u to state v is written $\langle u \to v \rangle$. Additionally, each path has a start state, which is of use primarily when the sequence of transitions is empty.

Our definitions are simplified by postulating a special state $\Lambda \notin U$, a "block" state. Unlike Pratt's Λ, our Λ can never actually be entered by a process. The role of Λ is explained in detail under blocking below.

Formally, the set of paths $\Psi(U)$ over U and the set of processes $\Pi(U)$ over U, where U is a countable set of states, are defined as

$$\Psi(U) = U \times (U \times U \cup \langle \Lambda \to \Lambda \rangle)^{*+\omega}$$

with the condition that if $(u, \langle v \to w \rangle \sigma) \in \Psi(U)$ then either $v=u$ or $v=\Lambda$.

$$\Pi(U) = P(\Psi(U)).$$

$S^{*+\omega}$ denotes finite and infinite sequences over S, and P denotes powerset. Some other useful definitions are

as follows:

Let $\psi = (u,\sigma)$ and $\psi' = (u',\sigma')$ be paths.

$\ell(\psi)$ = the number of transitions in σ (possibly ω).

start$(\psi) = u$.

$$\text{end}(\psi) = \begin{cases} u \text{ if } \sigma=\lambda, \\ w \text{ if } \sigma=\tau\langle v\to w\rangle, \\ \text{undefined if } \ell(\psi)=\omega. \end{cases}$$

ψ is a prefix of ψ' if σ is a prefix of σ', and $u=u'$.

The concatenation $\psi\cdot\psi'$ of ψ and ψ' is defined when $\ell(\psi')>0$.

$$\psi\cdot\psi' = \begin{cases} \psi & \text{if } \ell(\psi) = \omega \\ (u,\ \sigma\cdot\sigma') & \text{if } \ell(\psi) < \omega \end{cases}$$

The only restrictions on paths are that 1) Λ appear only in the transition $\langle\Lambda\to\Lambda\rangle$, and 2) the start state be the same as the first state in the first transition (or Λ). For example, $\psi = (u, \langle u\to w\rangle\langle y\to z\rangle)$ is an acceptable path, even when $w \neq y$. Path ψ represents a computation sequence which reaches state w, then moves from state y to state z, an impossibility. There are reasons for accepting such absurd paths. One is that some concurrent process, to be added later, could in fact make the phantom transition from w to y. Another reason is discussed below. We say that a path ψ is <u>legal</u> provided

$\psi = (u,\sigma\langle v\to w\rangle\langle y\to z\rangle\sigma') \Rightarrow w=y$.

The stages $S(U)$ are the finite legal paths over U. If π is a process, the set $pre(\pi)$ is the set of all stages which are prefixes of members of π.

Transition sequences have some advantages over ordinary computation sequences (sequences of states). One is that the concurrent execution of two processes can be defined simply as the shuffle of the transition sequences associated with each process. The same is not true for state sequences, for they don't retain enough information. Another advantage is that blocking, an important notion of concurrent processes, is readily defined in terms of transition sequences. A third advantage is that transitions can be labeled, so that it is possible to tell which process makes a given transition. Such labels are imperative if we wish to know if a given path is fair, in the sense that each process makes infinitely many transitions. Labels are discussed further in Chapter 5.

As can be seen from the lack of restrictions on processes, processes are nondeterministic. A process may have any number of paths whose initial state is u. Moreover, a process may contain any number of paths, all with prefix ψ. Intuitively, that means that, after running for a while and reaching stage ψ, there are many possible ways in which the process might continue. Processes may exhibit infinite nondeterminism, which means that, even when the set U is finite, and ignoring blocked paths,

a given process π might not be the set of paths in any finite branching tree. Infinite nondeterminism is required to represent several processes running concurrently, even when each component process is treelike (see [LF79]), provided the concurrency operation obeys the finite delay property: No component of a concurrent system which is ready to execute a transition infinitely often is forever denied executing a transition. A simple program which exhibits infinite nondeterminism is

(while x=0 do noop)//(x:=1).

The first component may run arbitrarily much faster than the second process, but not infinitely much faster. Hence, assuming x=0 at the start, the while loop may be executed any finite number of times, but not infinitely many times. We will find that infinite nondeterminism has special significance in both GPL and MPL, though in opposite ways. Our decision method for MPL makes use of processes which exhibit infinite nondeterminism, while that for GPL cannot deal with such processes.

Blocking

Every path has three possible fates; it may terminate, run forever or block. A terminating path is a finite legal path. Infinite legal paths run forever. And a blocking path is an illegal path, whether finite or

infinite. For example, let u and v be distinct states.

$(u, \langle \Lambda \to \Lambda \rangle)$

$(u, \langle u \to v \rangle \langle u \to v \rangle)$,

and $(u, \langle u \to v \rangle \langle \Lambda \to \Lambda \rangle)$

are all blocked paths. The transition $\langle u \to v \rangle$ cannot be executed unless the process is in state u. Note that the transition $\langle \Lambda \to \Lambda \rangle$ can never be executed on any path, for no path may begin in state Λ, and no transition of the form $\langle u \to \Lambda \rangle$, for $u \neq \Lambda$, is permitted. Thus $\langle \Lambda \to \Lambda \rangle$ is a "block marker." It is convenient to have such a marker which must always cause a block.

Our notion of blocking may be different from the reader's. In our notion, a block in a path merely means that the rest of the path is nonsense, suggesting that some other path be taken. Consider the program

while true do nothing.

In terms of PDL programs, defined shortly, "while true do nothing" is written

(true?)*; false?.

Suppose U contains a single state u. Then by the definition in the next section, (true?)*; false? represents the process $\pi = \{(u, \langle u \to u \rangle^k \langle \Lambda \to \Lambda \rangle) : k \geq 0\} \cup \{(u, \langle u \to u \rangle^\omega)\}$. Almost all of the paths block. But every stage $(u, \langle u \to u \rangle^k)$ is a prefix of some path which does not block at that

stage. Imagine an interpreter executing π. The interpreter must make nondeterministic choices. The choices can be made by choosing a path, say $(u, \langle u \to u \rangle^k \langle \Lambda \to \Lambda \rangle)$. After executing $\langle u \to u \rangle^k$, the interpreter encounters the transition $\langle \Lambda \to \Lambda \rangle$, which it cannot execute. Rather than giving up, the interpreter can choose a new path which has $\langle u \to u \rangle^k$ as a prefix, and so might just as well have been the chosen path. In fact, the interpreter can always find a path in π along which it can continue.

The interpreter (or "oracle," since it makes "correct" nondeterministic choices) just described is not built into processes in any sense. Rather, the statements which we make about processes can be looked at as having the form "when π is evaluated by a smart interpreter (one which tries other paths when a block is encountered on one path), then π obeys property p." For example, if we want to state that π cannot block, we do not say that π contains no blocked paths, but instead say that every legal prefix of every path in π is a proper prefix of some legal prefix of some (possibly different) member of π. The formula itself specifies the degree of wisdom of the oracle.

Reif and Peterson [RP80] carry the ability to specify the behavior of an oracle even further. In their logic, a formula can call for an oracle which is benevolent with respect to choices made by some components, and malicious with respect to choices made by others. Generally, it is

conceivable that some sort of "oracle specifier" could be added to the box operator of MPL (see Chapter 4), restricting the range of quantification over paths. We do not consider oracle specifiers in this work.

1.2. Programs

We use Propositional Dynamic Logic program syntax for our programs, with the addition of a concurrency operator. Thus concurrent programs are statically created, as in [LF79, OG76]. We do not make any provision for running arbitrarily many copies of a program in parallel, as in [S78]. PDL programs are particularly easy to give a semantics for. Also, in Chapter 2 we choose PDL as a termination logic framework, making PDL programs the most natural to use for our other logics. For the semantics of programs, we use processes. Program α represents process $\pi(\alpha)$. Basic programs are just symbols from a set Σ_0, and are given interpretation $\pi_0 : \Sigma_0 \rightarrow \Pi(U)$.

We place some restrictions on the processes represented by programs.

1. $\pi(\alpha)$ must not contain any paths of length zero. Each transition represents one unit of time. If α can completely execute in zero units of time, then α^* can execute infinitely many times in zero units of time, an undesirable situation.

2. For every state u, $\pi(\alpha)$ contains at least one

path starting at u. This is really no restriction, since $\pi(\alpha)$ may contain only the path $(u, \langle\Lambda\rightarrow\Lambda\rangle)$, which blocks without doing anything at all. This is mainly a technical restriction, making definitions slightly easier.

The syntax and semantics π:programs$\rightarrow\Pi(U)$ of programs is given below. Let α and β be programs.

1. Any basic program is a program, with $\pi(A) = \pi_0(A)$.

2. $\alpha \cup \beta$ is a program. $\alpha \cup \beta$ means "nondeterministically choose to run either α or β." $\pi(\alpha \cup \beta) = \pi(\alpha) \cup \pi(\beta)$.

3. $\alpha;\beta$ is a program. $\alpha;\beta$ means "run α, followed by β." $\pi(\alpha;\beta) = \pi(\alpha)\cdot\pi(\beta)$ (concatenation of processes).

4. α^* is a program. α^* means "run α any number (possibly ω) of times, the choice being made nondeterministacally. $\pi(\alpha^*) = \pi(\alpha)^{*+\omega}$.

5. $\alpha//\beta$ is a program. $\alpha//\beta$ means "run α and β in quasi-parallel." $\pi(\alpha//\beta)$ is the smallest set which satisfies the following:

Suppose $(u, \sigma_1\cdot\sigma_2 \ldots) \in \pi(\alpha)$ and

$(v, \tau_1\cdot\tau_2 \ldots) \in \pi(\beta)$,

where σ_1 and τ_1 are nonempty, and σ_i and τ_i are either empty or finite or infinite for $i>1$.

Then

$(u,\sigma_1\cdot\tau_1\cdot\sigma_2\cdot\tau_2 \ldots) \in \pi(\alpha//\beta)$

and $(v,\tau_1\cdot\sigma_1\cdot\tau_2\cdot\sigma_2 \ldots) \in \pi(\alpha//\beta)$.

By our definition, $\alpha//\beta$ does not obey the finite delay property. It is possible to define $\alpha//\beta$ with finite delay, by insisting that each τ_i and σ_i be finite. We will sometimes consider this alternate definition of // in the work that follows.

6. If p is a formula of a certain type, which may be different for each logic, then p? is a program. The truth value of p must depend only on a state. When p is true, p? executes a null transition. When p is false, p? cannot execute, and so must block.

$$\pi(p?) = \{(u,\langle u \rightarrow u\rangle): p \text{ holds in state } u\}$$
$$\cup \{(u,\langle \Lambda \rightarrow \Lambda\rangle): p \text{ does not hold in state } u\}$$

1.3. Truth in GPL and MPL

The logics GPL and MPL are defined in terms of a Kripke style truth value semantics. A structure $A = (U,\pi,\Phi_o,\phi_o)$ consists of a set U of states, a process π in $\Pi(U)$, a set Φ_o of basic formulas or predicates, and an interpretation $\phi_o:\Phi_o \rightarrow P(U)$ which assigns to P the set of states where P holds. The truth value of a formula depends on a structure, as well as some additional parameters, which differ slightly between GPL and MPL, mainly because a GPL formula may have many free variables whose values must be specified. An environment, or model, contains all of the information necessary to determine the truth value of a formula. For each logic, a relation $E \models P$, read

"E satisfies P," between environments and formulas is defined. We say that a formula p is satisfiable if there is some environment E which satisfies P. We say that p is valid if $\sim p$ is not satisfiable, i.e., if p is satisfied by every environment.

Chapter 2

Boolean Variables in Propositional Dynamic Logic

A reasonable first approach to dealing with concurrent programs is simply to add a concurrency operator to an established sequential program logic, such as Propositional Dynamic Logic. The semantics of programs may have to be changed to be able to define the concurrent execution of two programs. Such a logic would be suitable at least for describing termination properties of concurrent programs. (As we show below, it is capable of much more.)

PDL programs are close to regular expressions. It is well known that the shuffle of two regular sets is a regular set [GS65]. Hence it would seem reasonable that, at least in some cases, the concurrency operator could be expressed in terms of $\cup$, ; and *. That is the case when basic programs must be indivisible, i.e., every path in $\pi_0(A)$ must have length one.

For the rest of this chapter we adopt the convention that basic programs are indivisible. In this view, basic programs represent low level instructions, which are executed in a single step, as opposed to more complex programs. The restriction to indivisible basic programs greatly simplifies the study of concurrency, by allowing us to know just how programs can be interleaved. If A

and B are two non-indivisible basic programs, it is difficult to know what A//B will do, given only the behavior of A and B individually. We are not the first to restrict basic programs to indivisible actions (see e.g. [OG76, Pn77, RP80, N79]).

Infinite paths in $\pi(\alpha)$ can have no bearing on the truth of $[\alpha]p$, which only states that the finite legal paths end on a state satisfying p. So eliminating all infinite paths from all processes can have no effect on the truth of any PDL formula. Consequently, the two possible definitions of $PDL_{//}$, one with finite delay and the other without finite delay, must in fact be identical. Until we leave the realm of PDL, we can ignore the question of finite or infinite delay.

Although concurrency can be eliminated from PDL programs, the elimination is costly, the best known method causing a double exponential length blowup. We certainly would hope for a better method of handling concurrency than the brute force method of considering all possible ways of interleaving programs. Such a method does exist. Suppose we introduce into PDL auxiliary variables, whose values can be assigned and tested without affecting in any way the behavior of basic programs. Those variables could be used to efficiently write an "interpreter," which evaluates a concurrent program. By storing one or more program counters in variables, the interpreter can

remember where one or more programs are at each instant. The auxiliary variables can also be used to help express properties other than simple termination properties of programs. For example, to state that p holds throughout the execution of α, we simply write $[I_\alpha]p$, where I_α is an interpreter for α which may halt at *any time during* the evaluation of α.

Below we define an extension B-PDL of PDL which includes Boolean variables. We list a number of concepts which B-PDL can simulate. We also show that every B-PDL formula is equivalent, in a sense defined precisely later, to some PDL formula. Consequently, any concept which can be expressed in B-PDL can also be expressed (albeit less concisely) in PDL. We prove upper and lower bounds on the time complexity of the satisfiability problem for B-PDL. A related upper bound naturally applies to any logic which can be efficiently simulated by B-PDL.

2.1. B-PDL

We begin by giving formal definitions of sequential PDL and B-PDL. Because the box operator of PDL only looks at the first and last states of a path, we can simplify the semantics of programs, letting each path consist only of a start state and a final state. A program represents a set of such paths, which is just a binary relation over states.

After defining B-PDL, we show informally that B-PDL can efficiently simulate certain notions, such as concurrency, which really require that programs represent sets of paths rather than binary relations. Given the definition in Chapter 1 of $\pi(\alpha)$, the reader should have little difficulty in extending our relational definition of B-PDL to a definition based on processes. A formal proof that concurrency can be eliminated from B-PDL formulas naturally must be carried out in a version of B-PDL which includes concurrency, and whose programs represent processes rather than relations.

The following definition of PDL is taken from [FL79]. A PDL structure $A = (U, \Phi_0, \Sigma_0, \phi_0, \rho_0)$ consists of

U = a set of states;

Φ_0 = a set of basic formulas;

Σ_0 = a set of basic programs;

ϕ_0: $\Phi_0 \rightarrow P(U)$, assigning to each basic formula the state where it holds;

ρ_0: $\Sigma_0 \rightarrow P(U \times U)$, assigning to each basic program a binary relation over U.

The programs Σ, formulas Φ, and their associated semantics ρ: $\Sigma \rightarrow P(U \times U)$ and ϕ: $\Phi \rightarrow P(U)$ are defined inductively as follows.

Programs

1. $A \in \Sigma_0$ is a program with $\rho(A) = \rho_0(A)$.
2. Let $\alpha, \beta \in \Sigma$, $p \in \Phi$.
   a) $\alpha \cup \beta \in \Sigma$, $\rho(\alpha \cup \beta) = \rho(\alpha) \cup \rho(\beta)$;
   b) $\alpha;\beta \in \Sigma$, $\rho(\alpha;\beta) = \rho(\alpha) \cdot \rho(\beta)$ (composition of relations);
   c) $\alpha^* \in \Sigma$, $\rho(\alpha^*) = \rho(\alpha)^*$ (reflexive transitive closure of a relation);
   d) $p? \in \Sigma$, $\rho(p?) = \{(u,u): u \in \phi(p)\}$.

Formulas

1. $P \in \Phi_0$ is a formula, with $\phi(P) = \phi_0(P)$.
2. Let $p,q \in \Phi$, $\alpha \in \Sigma$.
   a) $\sim p \in \Phi$, $\phi(\sim p) = P(U) - \phi(p)$.
   b) $p \vee q \in \Phi$, $\phi(p \vee q) = \phi(p) \cup \phi(q)$.
   c) $\langle\alpha\rangle p \in \Phi$, $\phi(\langle\alpha\rangle p) = \{u: \exists v((u,v) \in \rho(\alpha)$ and $v \in \phi(p))\}$.

   ($[\alpha]p$ is defined as $\sim\langle\alpha\rangle\sim p$.)

For a thorough discussion of PDL, see [FL79]. We generally write $u \Vdash p$ for $u \in \phi(p)$. The symbols $\wedge$, $\supset$, $\equiv$, etc. have their usual meanings. We remark that the PDL program constructs can express the usual if-then-else and while-do constructs, as

if p then a else b = $(p?;a) \cup (\sim p?;b)$,

while p do a = $(p?;a)^*; \sim p?$.

We proceed now to B-PDL. A B-PDL structure contains, in addition to all of the members of a PDL structure, a set V of Boolean variables. If x is a Boolean variable, then x is a formula, and x↑(set x) and x↓(reset x) are programs. The truth of a formula depends not only on a state u, but also on a set s containing the Boolean variables which are true. Programs of B-PDL represent relations over $U \times P(V)$, with the basic programs altering only the first component, and the set and reset programs altering only the second component. Using separate components achieves the desired independence of variable actions and program actions which is necessary to write the sort of interpreter described earlier. The sets Σ_B of B-PDL programs and Φ_B of B-PDL formulas, along with their respective semantics $\rho_B: \Sigma_B \to P((U \times P(V))^2)$ and $\phi_B: \Phi_B \to P(U \times P(V))$ are defined inductively below.

Programs

1. $A \in \Sigma_0$ is a program in Σ_B with
   $\rho_B(A) = \{((u,s), (v,s)): (u,v) \in \rho_0(A), s \subseteq V\}$.

2. Let $x \in V$.
   a) $x\uparrow \in \Sigma_B$, $\rho_B(x\uparrow) = \{((u,s), (u,s')): s' = s \cup \{x\}\}$;
   b) $x\downarrow \in \Sigma_B$, $\rho_B(x\downarrow) = \{((u,s), (u,s')): s' = s - \{x\}\}$.

3. $\alpha \cup \beta$, $\alpha;\beta$ and α^* are defined exactly as for PDL. If p is in Φ_B, then p? is a program, with $\rho_B(p?) = \{((u,s), (u,s)): (u,s) \in \phi_B(p)\}$.

Formulas

1. $P \in \Phi_0$ is a formula, with $\phi_B(P) = \phi_0(P) \times P(V)$.

2. $x \in V$ is a formula, with $\phi_B(x) = U \times \{s \subseteq V: x \in s\}$.

3. Let $p,q \in \Phi_B$, $\alpha \in \Sigma_B$.
   a) $\sim p \in \Phi_B$, $\phi_B(\sim p) = U \times P(V) - \phi_B(p)$;
   b) $p \vee q \in \Phi_B$, $\phi_B(p \vee q) = \phi_B(p) \cup \phi_B(q)$;
   c) $\langle\alpha\rangle p \in \Phi_B$, $\phi_B(\langle\alpha\rangle p) = \{(u,s): (\exists v \in U, t \subseteq V)((u,s),(v,t)) \in \rho_B(\alpha)$ and $(v,t) \in \phi_B(p)\}$.

We write $u,s \models p$ for $(u,s) \in \phi_B(p)$. Below is a list of examples demonstrating the power of B-PDL.

1. Integers in the range of 0 to 2^n-1 can be represented using n Boolean variables. It is routine to write a program of length 0(n) which adds or subtracts two such integers, or tests them for equality, or determines which is larger. Bounded quantification over the range $[0, 2^n-1]$ is expressed as $\forall x = [x \leftarrow \text{random}]$, where $x \leftarrow \text{random} = (x_1\uparrow \cup x_1\downarrow); \ldots, (x_n\uparrow \cup x_n\downarrow)$.

2. The program $\alpha^n = \alpha; \ldots; \alpha$(n times) can be abbreviated using Boolean variables, representing integers up to n, as follows:

I ← 0;

while I ≠ n do (α; I ← I + 1).

This program has length ℓ(α) + 0(log n), which may be considerably shorter than nℓ(α). The programs x↑, x↓, x? and ∿x? used in the while-loop cannot affect the state component of (u,s), and so cannot affect the running of α, so long as none of the variables used to simulate I appears in α.

3. Using small integers, we can convert a flowchart of n boxes, whose boxes contain basic programs and tests, to a length 0(n log n + length of all tests) B-PDL program. The program has the form S; $(\bigcup_i T_i)^*$; F?, where S sets a counter to the start box; T_i tests if the counter is i, and if so performs the action in box i, then setting i to the number of the next box; and F tests for a final box. The length of the Boolean variable simulation of a flowchart is generally much shorter than the standard PDL simulation, which must be exponential in n in the worst case. Decidability of PDL with flowchart programs follows from the decidability of B-PDL. Pratt [Pr80] gives a single exponential time decision method for PDL with flowcharts, which is slightly better than that obtainable from B-PDL.

4. Any length n program can be changed to the form S; $(\bigcup_i T_i)^*$; F? of example 3, with a factor of c log n length increase for some constant c. Simulations below

make use of this program translation.

5. Subroutine calls to bounded depth, with small integer parameters, can be simulated in the obvious way.

6. Concurrency can be simulated by allowing more than one counter to be active at a time. Each pass selects which counter is to be used, then uses it in the usual way. The nondeterminism inherent in concurrent programs is simulated by the nondeterminism which is built into PDL. This simulation treats basic programs as indivisible actions.

7. A kind of labeling already is in use. It is a simple matter to allow syntactic labels in programs, and to test for being at a given label, or in a given region (using special binary encodings, which allow for testing only the most significant bits). We can also test which program made the last transition, using a backup counter.

8. Global invariance of p over α can be expressed in B-PDL. If $S; (\bigcup_i T_i)^*;F?$ is an equivalent program to α, then "p holds throughout the execution of α" is expressed as $[S; (\bigcup_i T_i)^*]p$, where the termination test has been omitted. T_i executes a single step of α.

9. We can test whether every possible execution sequence of a program must obey p <u>while</u> q, which says that, as long as p holds, q holds. If α is represented by $S; (\bigcup_i T_i)^*;F?$, then α satisfies p <u>while</u> q provided

$$[S;(q?;\ \bigcup_i T_i)^*](q \supset p).$$

The two occurrences of q can be reduced to one by the use of more Boolean variables.

10. B-PDL simulates "p holds at the next instant" by only running $\bigcup_i T_i$ once. The operator "until" of [GPSS80] is shown in Chapter 4 to be expressible in terms of while and "next." Hence in B-PDL we can express that every path of α satisfies p until q. However, B-PDL cannot simulate nexted whiles or untils, at least under the meaning of [GPSS80]. For each time an "until" simulation is done, B-PDL requantifies the path in question. That is, B-PDL can only simulate branching time modalities, in the sense of [L80].

11. Using interpreters, it is possible to "remember" as program counter from one modality to the next. Consider the statement "α preserves p," i.e., "α never changes p from true to false." Letting $S;(\bigcup_i T_i)^*;F?$ be an interpreter for α, "α preserves p" can be expressed in B-PDL as $[S][(\bigcup_i T_i)^*](p \supset [(\bigcup_i T_i)^*]p)$. In words, after α is started and run for some number of steps, if p holds, then continuing α for any more steps must lead to a state where p holds. $[S][(\bigcup_i T_i)^*]p$ is just an expression of global invariance. Pratt's process logic [Pr78] includes a global invariance operator $\{\alpha\}p$. But $\{\alpha\}(p \supset \{\alpha\}p)$ does not express "α preserves p," for the nested $\{\alpha\}p$

restarts α. Pratt's logic, as well as Parikh's SOAPL [Pa78] and Nishimura's process logic [N79], have no obvious means of expressing that α should take up where it left off.

In $[S][(\bigcup_i T_i)^*](p \supset [(\bigcup_i T_i)^*]p)$, we must write $(\bigcup_i T_i)^*$ twice. It would seem more reasonable to invent a form such as $\alpha\cdot[](p \supset []p)$, where $\alpha\cdot$ means $[S]$, and determines all T_i, and $(\bigcup_i T_i)^*$, or its semantic equivalent, is implicit in each box. Such a form is introduced in Chapter 5.

B-PDL has been shown to be a rich language, and merits study. B-PDL is also interesting in its own right as PDL with very simple assignment programs. The remainder of this chapter is devoted to proving results about B-PDL. We begin by proving that Boolean variables can be eliminated from B-PDL formulas. We then give a characterization of B-PDL in terms of PDL, and using it, show that the satisfiability problem for B-PDL is decidable in nondeterministic time c^{n3^m}, where n is the length of a formula, and m is the number of distinct Boolean variables which it contains. Lastly, we prove a deterministic time d^{n2^m} lower bound on SAT(B-PDL).

2.2. Equivalence of B-PDL and PDL

Since B-PDL formulas can reference Boolean variables, it is clear that PDL cannot strictly express as much as

B-PDL. But if the initial values of all of the Boolean variables are fixed, then we can show that PDL can express just as much as B-PDL. Precisely, for every set s of Boolean variables, there is a map T_s from B-PDL formulas into PDL formulas for which $u,s \models p$ iff $u \models T_s(p)$ for every state u , and every B-PDL formula p.

It is easy to see how to translate a formula of the form $[\alpha]p$ to PDL, where α may contain programs of the form $x\uparrow$, $x\downarrow$, $x?$ and $\sim x?$, but not arbitrary tests. Begin by constructing a nondeterministic finite automaton F equivalent to regular expression α, treating $x\uparrow$, $x\downarrow$, $x?$ and $\sim x?$ as symbols of the alphabet. Next, if α contains m distinct Boolean variables, make 2^m copies of F, one for each different subset of the Boolean variables. Arcs labeled $x\uparrow$ and $x\downarrow$ are eliminated by turning them into λ-arcs between copies of F. $x?$ arcs are either turned into λ-arcs or are erased. Finally, the resulting finite automaton is converted into a regular expression. We see that, if α is of length n, then the program α' which we construct from α has length c^{n2^m} for some c. The upper bound c^{n2^m} is very poor when m=0, in which case our translation causes an exponential blowup when no change at all is necessary. Nevertheless, when m is large, we conjecture that the bound is tight. A double exponential lower bound is proved in [A80] on the length blowup incurred in translating Boolean variable regular expressions into

ordinary regular expressions. Hence there is a B-PDL formula $[\alpha]P$ which is not equivalent to any short PDL formula of the form $[\alpha']P$. That does not preclude the possibility of a short PDL formula equivalent to $[\alpha]P$ which is of some altogether different form. The best lower bound we can prove is single exponential, resulting from translating $[A^k]p$ to $[A; \ldots; A]p$.

We now describe the translation T_s.

Theorem 2.1. Let p be a B-PDL formula of length n, containing m distinct Boolean variables $x_1, \ldots, x_m$, and let s be a subset of $\{x_1, \ldots, x_m\}$. There is a map T_s: $\Phi_B \to \Phi$ such that for every state u, $u,s \models p$ iff $u \models T_s(p)$, and $T_s(p)$ has length at most $O(n + d^{n2^m})$, for some d.

Proof. Let $i_{(k)}$ be the kth bit, numbered left to right, of the binary representation of i. Let t_i be the conjunction of Boolean variables and their negations which is true iff $x_1 \ldots x_m$ is the binary representation of i. Let s_i be the program which sets $x_1 \ldots x_m$ to the integer i. The vector $[p_0, \ldots, p_{2^m-1}]$ of formulas represents the formula $\bigvee_{i=0}^{2^m-1} (t_i \wedge p_i)$. The matrix $[\alpha_{ij}]_{2^m \times 2^m}$ represents the program $\bigcup_{i=0}^{2^m-1} \bigcup_{j=0}^{2^m-1} (t_i?; \alpha_{ij}; s_j)$. Define the length of a vector or matrix as the length of its longest component.

We inductively define translation T which maps every B-PDL formula into a vector formula whose components contain no Boolean variables, and which maps every B-PDL program into a matrix program each of whose components contains no Boolean variables. Simultaneously we prove that $p \equiv T(p)$ and $\rho(\alpha) = \rho(T(\alpha))$ in every structure. $T_s(p)$ is just the component of $T(p)$ in the position corresponding to s. Let $\ell = 2^m-1$.

P. $T(P) = [P, \ldots, P]$, and $P \equiv T(P)$ is trivial.

x_k. $T(x_k) = [q_o, \ldots, q_\ell]$, where

$$q_i = \begin{cases} \underline{\text{true}} \text{ if } i_{(k)} = 1 \\ \underline{\text{false}} \text{ if } i_{(k)} = 0 \end{cases}$$

The proof that $x_k \equiv T(x_k)$ is trivial.

$\sim p$. Let $T(p) = [p_o, \ldots, p_\ell]$. Then

$$T(\sim p) = [\sim p_o, \ldots, \sim p_\ell].$$

We prove by induction on m that $\sim[p_o, \ldots, p_\ell] \equiv [\sim p_o, \ldots, \sim p_\ell]$.

Basis. (m=0). $\sim[p_o] \equiv [\sim p_o]$.

Induction. Let I_o and I_1 be disjoint index sets such that $I_o \cup I_1 = \{0, \ldots, 2^m-1\}$, and $i \in I_o$ iff $i_{(m)} = 0$. Let $t_i \equiv (\sim)x_m \wedge t_i'$.

$$\sim p \equiv \sim[p_o, \ldots, p_{2^m-1}]$$

$$\equiv \bigvee_{i=1}^{2^m-1} t_i \wedge p_i,$$

$$\equiv \sim((\sim x_m \wedge \bigvee_{i\in I_o} (t_i' \wedge p_i)) \vee (x_m \wedge \bigvee_{i\in I_1} (t_i' \wedge p_i))),$$

$$\equiv (x_m \vee \sim\bigvee_{i\in I_o} (t_i' \wedge p_i)) \wedge (\sim x_m \vee \sim\bigvee_{i\in I_1} (t_i' \wedge p_i)).$$

Each index set I_o and I_1 covers all possible subsets of $x_1, \ldots, x_{m-1}$, so $\bigvee_{i\in I_o} t_i' \wedge p_i$ and $\bigvee_{i\in I_1} t_i' \wedge p_i$ are vector formulas. By induction

$$\sim p \equiv (x_m \vee \bigvee_{i\in I_o} (t_i' \wedge \sim p_i)) \wedge (\sim x_m \vee \bigvee_{i\in I_1} (t_i' \wedge \sim p_i)).$$

By the tautology $(a \vee b) \wedge (\sim a \vee c) \equiv (\sim a \wedge b) \vee (a \wedge c)$,

$$\sim p \equiv (\sim x_m \wedge \bigvee_{i\in I_o} (t_i' \wedge \sim p_i)) \vee (x_m \wedge \bigvee_{i\in I_o} (t_i' \wedge \sim p_i)),$$

$$\equiv [\sim p_o, \ldots, \sim p_{2^m-1}]$$

by the choice of the index sets.

$\underline{p \vee q}$. Let $T(p) = [p_o, \ldots, p_\ell]$ and $T(q) = [q_o, \ldots, q_\ell]$. Then $T(p \vee q) = [p_o \vee q_o, \ldots, p_\ell \vee q_\ell]$, which is easily shown to be correct.

$\underline{\langle\alpha\rangle p}$. Let $T(p) = [p_o, \ldots, p_\ell]$ and $T(\alpha) = [\alpha_{ij}]_{2^m \times 2^m}$. then

$$T([\alpha]p) = [\alpha_{ij}] \begin{pmatrix} \Diamond \\ \vee \end{pmatrix} \begin{bmatrix} p_o \\ \vdots \\ p_\ell \end{bmatrix},$$

the matrix-vector product with $+ = \vee$ and $\cdot = \Diamond$, where $\alpha \Diamond p = \langle\alpha\rangle p$. It is straightforward to show that $T(\langle\alpha\rangle p) \equiv \langle\alpha\rangle p$.

T is defined below for programs. None of the cases presents any difficulty, and correctness proofs are omitted.

A.

$$T(A) = \begin{bmatrix} A & & \emptyset \\ & \ddots & \\ \emptyset & & A \end{bmatrix}, \text{ where } \emptyset = \text{false?}.$$

$x_k\uparrow$. $T(x_k\uparrow) = [\beta_{ij}]_{2^m \times 2^m}$, where

$$\beta_{ij} = \begin{cases} \text{true? if } j_{(k)} = 1 \text{ and} \\ \qquad i_{(h)} = j_{(h)} \text{ for all } h \neq k, \\ \text{false? otherwise.} \end{cases}$$

$x_k\downarrow$. Similar to $x_k\uparrow$.

p?. Let $T(p) = [p_o, \ldots, p_\ell]$. Then

$$T(p)? = \begin{bmatrix} p_o? & & \emptyset \\ & \ddots & \\ \emptyset & & p_\ell? \end{bmatrix}, \text{ where } \emptyset = \text{false?}.$$

$\alpha \cup \beta$. $T(\alpha \cup \beta) = T(\alpha) \cup T(\beta)$ (componentwise union).

$\alpha;\beta$. $T(\alpha;\beta) = T(\alpha) \binom{\cdot}{\cup} T(\beta)$ (matrix multiplication).

α^*. $T(\alpha^*) = T(\alpha)^*$, the reflexive transitive closure of $T(\alpha)$ with respect to the $\binom{\cdot}{\cup}$ product.

Length of T(p). The length increase due to the transitive closure for $T(\alpha^*)$ dominates the others by far, provided the usual algorithm is used to take matrix products. When k is a power of 2, the transitive closure of a k × k matrix can be computed recursively by dividing

the matrix into four square submatrices, and applying the formula (see [AHU74, p. 205]).

$$M^* = \begin{bmatrix} A & B \\ C & D \end{bmatrix}^* = \begin{bmatrix} (A + BD^*C)^* & (D + CA^*B)^*CA^* \\ D^*C(A + BD^*C)^* & (D + CA^*B)^* \end{bmatrix}.$$

Let $s(k,\ell)$ be the length of $M^*_{k\times k}$ where ℓ is the length of M (i.e., the length of its longest component). A simple substitution argument reveals that $s(k,\ell) \leq \ell s(k,1)$. Let $s(k) = s(k,1)$. The length of the product of two $k\times k$ matrices of length ℓ_1 and ℓ_2, taken by the standard multiplication algorithm, is $O(k(\ell_1 + \ell_2))$. That fact and the formula for M* lead to

$$s(k) \leq O(k^2) + O(k^4)s(k/2) + O(k^4)s(k/2)^2 \text{ for } k > 1,$$

from which it can be shown that

$$s(k) = O(c^k/k^4) \qquad \text{for some } c,$$
$$\leq d^k \qquad \text{for some } d.$$

Claim. Length (T(α)) and length (T(p)) are both $O(d^{n2^m})$, where n is the length of α (or p), and m>0 is the number of distinct Boolean variables in α (or p).

Proof. By induction on the length of α or p. Technically, we must consider each case. Since α* dominates all others, we show the proof only for α*.

$$\text{length } T(\alpha^*) \leq s(2^m, \text{ length } T(\alpha)),$$
$$\leq \text{length } T(\alpha)\cdot s(2^m),$$
$$\leq c\, d^{(n-1)2^m}\, d^{2^m} \qquad \text{by induction,}$$
$$= c\, d^{n2^m}.$$

The length part of theorem 2.1 follows from the claim, and separate analysis of the trivial case m=0. ∎

The test depth of a formula is defined as the depth of nesting of the "?" operator, with formulas of test depth zero containing no tests. Let PDL_n be the PDL formulas with test depth at most n. Berman and Peterson [BP78] have shown that PDL_{n+1} is strictly more expressive than PDL_n. We see by inspection that our translation from B-PDL into PDL does not increase test depth. Hence $B\text{-}PDL_{n+1}$ is more expressive than B-PDL. That contrasts with the case of star depth, any program being expressible with a single star by the use of Boolean variables (except where stars must be nested solely as a consequence of test nesting). Cohen [Co70] has shown that regular expressions require large star depth for full expressive power. We conjecture that the same holds for PDL.

2.3. A characterization of B-PDL

Rather than defining B-PDL separately from PDL, it is possible to define B-PDL as ordinary PDL, subject to certain axioms concerning the behavior of $x\uparrow$ and $x\downarrow$. Axioms B1-B4 completely define B-PDL. While B1-B4 represent a step in the direction of obtaining a complete axiomatization of B-PDL, B1-B4 are not the usual type of axiom, being inexpressible in PDL. Consequently B1-B4 can't be directly

used to either decide or prove the validity of a B-PDL formula by falling back on the methods used for PDL. Nevertheless, B1-B4 can be used to extend theorems about PDL to B-PDL. In B1-B4, $x\uparrow$ and $x\downarrow$ are considered special basic programs associated with the basic formula x. Those basic formulas which have set and reset programs associated with them are called Boolean variables. A PDL structure which satisfies B1-B4 is called Boolean with respect to the set V of Boolean variables and the map which assigns $x\uparrow$ and $x\downarrow$ to x, or simply Boolean when V and the map are understood.

B1. The following hold at every state, for every Boolean variable $x \in \Phi_0$, and every basic formula or Boolean variable $P \in \Phi_0 - \{x\}$.

a) $\langle x\uparrow\rangle$true,

b) $\langle x\uparrow\rangle$true,

c) $[x\uparrow]x$,

d) $[x\downarrow]\neg x$,

e) $P \supset [x\uparrow]P$,

f) $P \supset [x\downarrow]P$.

B2. a) $u \vdash x \Rightarrow ((u,v) \in \rho(x\uparrow) \Rightarrow u=v)$,

b) $u \vdash \sim x \Rightarrow ((u,v) \in \rho(x\downarrow) \Rightarrow u=v)$.

B3. a) $\rho(x\uparrow;x\downarrow) = \rho(x\downarrow)$,

b) $\rho(x\downarrow;x\uparrow) = \rho(x\uparrow)$.

B4. For all $A \in \Sigma_0 - \{x\uparrow, x\downarrow\}$,

a) $\rho(A;x\uparrow) = \rho(x\uparrow;A)$,

b) $\rho(A;x\downarrow) = \rho(x\downarrow;A)$.

B1 expresses the behavior of $x\uparrow$ and $x\downarrow$ relative to the basic formulas (including x), and requires no justification. It is clear that B1 alone cannot completely define B-PDL. For if it did, it would be possible to decide the satisfiability of any B-PDL formula p by merely conjoining appropriate instances of B1 to p, and testing whether the result is a satisfiable PDL formula, violating the lower bound on SAT(B-PDL) which is proved later. B2 and B3 are required to make a reduction of a Boolean PDL model isomorphic to a corresponding B-PDL model. Whether B2 and B3 are required to define SAT(B-PDL) is questionable. B4 is a statement of independence of $x\uparrow$ and $x\downarrow$ from every other basic program. It is the independent action of $x\uparrow$ and $A \notin \{x\uparrow, x\downarrow\}$ which is difficult to enforce using only expressions which can be written in PDL. The following consequences of B1-B4 will prove useful. Each is stated and proved only for $x\uparrow$, though dual statements for $x\downarrow$ also hold.

B5. If $u \models x$ then $(u,u) \in \rho(x\uparrow)$.

Proof. Immediate from B1(a), B2(a). ∎

B6. Suppose $u \models \sim x$. Then there is a $v \neq u$ such that $(u,v) \in \rho(x\uparrow)$ and $(v,u) \in \rho(x\downarrow)$.

Proof. By B5 $(u,u) \in \rho(x\downarrow) = \rho(x\uparrow;x\downarrow)$, which means there is a v such that $(u,v) \in \rho(x\uparrow)$ and $(v,u) \in \rho(x\downarrow)$. By B1(c), $v \models x$, so $u \neq v$. ∎

B7. (Determinism) For every u there is at most one v such that $(u,v) \in \rho(x\uparrow)$.

Proof. If $u \models x$, B7 follows immediately from B2. Suppose $u \models \sim x$. By B6, there is a v such that $(u,v) \in \rho(x\uparrow)$ and $(v,u) \in \rho(x\downarrow)$ and $v \models x$.

$(u,v') \in \rho(x\uparrow) \Rightarrow (v,v') \in \rho(x\downarrow;x\uparrow)$ by $(v,u) \in \rho(x\downarrow)$,

$\Rightarrow (v,v') \in \rho(x\uparrow)$ by B3,

$\Rightarrow v = v'$ by B2, since $v \models x$. ∎

B8. (Reversibility) If $(u,v) \in \rho(x\uparrow)$ and $u \neq v$ then $(v,u) \in \rho(x\downarrow)$.

Proof. Immediate from B6 and B7. ∎

B9. If $A \in \Sigma_0 - \{x\uparrow, x\downarrow\}$ then $x \supset [A]x$ and $\sim x \supset [A]\sim x$.

Proof. We prove $x \supset [A]x$ only.

$u \Vdash x \wedge (u,v) \in \rho(A) \Rightarrow (u,u) \in \rho(x\dagger) \wedge (u,v) \in \rho(A)$ by B5,

$\Rightarrow (u,v) \in \rho(x\dagger;A)$,

$\Rightarrow (u,v) \in \rho(A;x\dagger)$ by B4,

$\Rightarrow \exists v'(v',v) \in \rho(x\dagger)$,

$\Rightarrow v \Vdash x$ by B1. ∎

Theorem 2.2. Every B-PDL formula with Boolean variables V is satisfiable iff it is satisfiable by some Boolean PDL structure (i.e., one obeying B1-B4) with Boolean variables V.

Proof. ($\Rightarrow$) Given a B-PDL structure $\mathcal{B} = (U, \Phi_0, \Sigma_0, V, \phi_0, \rho_0)$ such that $\mathcal{B},u,s \Vdash p$, define a PDL structure $A = (U \times P(V), \Phi_0 \cup V, \Sigma_0 \cup \{x\dagger, x\downarrow : x \in v\}, \phi_0', p_0')$, where ϕ_0' and p_0' are defined in the obvious way. It is easy to verify that A satisfies B1-B4, and that $A,(u,s) \Vdash p$.

($\Leftarrow$) Suppose $A = (U, \Phi_0, \Sigma_0, \phi_0, \rho_0)$ obeys B1-B4, and $A,u \Vdash p$. Define the equivalence relation $\equiv$ over U by

$u \equiv v$ iff there is a sequence $d_1, \ldots, d_n$, $n \geq 0$, of set and reset programs such that $(u,v) \in \rho(d_1; \ldots; d_n)$.

$\equiv$ is symmetric by B8, and is clearly reflexive and transitive. Define B-PDL structure $\mathcal{B} = (\bar{U}, \Phi_0 - V, \Sigma_0 - \{x\dagger, x\downarrow : x \in V\}, V, \bar{\phi}_0, \bar{\rho}_0)$ by

$\bar{u} = \{v : u \equiv v\}$,

$\overline{U} = \{\overline{u}: u \in U\}$,

$\overline{\phi}_o(P) = \{\overline{u}: u \in \phi_o(P)\}$,

$\overline{\rho}_o(A) = \{(\overline{u},\overline{v}): (u,v) \in \rho(A)\}$.

Claim. Let $S_u = \{x \in V: u \vdash x\}$. For all p and α,

1) $A,u \vdash p$ iff $B,\overline{u},S_u \vdash p$,

2) $(u,v) \in \rho(\alpha)$ iff $((\overline{u},S_u),(\overline{v},S_v)) \in \overline{\rho}(\alpha)$.

Once the claim is proved, we are finished proving theorem 2.2. The claim is proved by simultaneous induction on formula and program length. We need a lemma.

Lemma 2.3. If $u \equiv v$ and $S_u = S_v$ then u=v.

Proof. Let R be a sequence of set and reset programs such that $(u,v) \in p(R)$. Using B3 and B4, R can be reduced to contain at most one set or reset program per variable. If $S_u=S_v$, the set or reset program for x in R cannot change the value of x. By repeated application of B2, u=v. ∎

Proof of Claim.

P $\in \phi_o$-V. $A,u \vdash P \Leftrightarrow u \in \phi_o(P)$,

$\Leftrightarrow \overline{u} \in \overline{\phi}_o(P)$ from B1(e,f),

$\Leftrightarrow B,\overline{u},S_u \vdash P$.

$x \in V$. $A,u \vdash x \Leftrightarrow u \in \phi_o(x)$,

$\Leftrightarrow x \in S_u$ by definition of S_u,

$\Leftrightarrow B,u,S_u \vdash x$.

$\therefore$ $\underline{\sim p,\ p \vee q}$. Trivial.

$\underline{\langle\alpha\rangle p}$. $A,u \models \langle\alpha\rangle p \Leftrightarrow \exists v((u,v) \in \rho(\alpha) \wedge A,v \models p)$,

(*) $\Leftrightarrow \exists v(((\bar{u},S_u),(\bar{v},S_v)) \in \bar{\rho}(\alpha)$
$\wedge\ B,\bar{v},S_v \models p)$

by induction,

$\Rightarrow B,\bar{u},S_u \models \langle\alpha\rangle p.$

Conversely,

$B,\bar{u},S_u \models \langle\alpha\rangle p \Rightarrow \exists v,s(((\bar{u},S_u),(\bar{v},s)) \in \bar{\rho}(\alpha)$
$\wedge\ B,\bar{v},s \models p).$

It is possible to find $v' \equiv v$ such that $S_{v'} = s$, by running appropriate set and reset programs from v,

$\Rightarrow \exists v'(((\bar{u},S_u),(\bar{v}',S_{v'})) \in \bar{\rho}(\alpha) \wedge B,\bar{v}',S_{v'} \models p)$,

$\Rightarrow A,u \models \langle\alpha\rangle p$ by (*).

$\underline{A \neq x\uparrow}$. $(u,v) \in \rho_o(A)$

$\Rightarrow ((\bar{u},S_u),(\bar{v},S_u)) \in \overline{\rho_o}(A)$ and $S_u = S_v$ by B9,

$\Rightarrow ((\bar{u},S_u),(\bar{v},S_v)) \in \overline{\rho_o}(A)$.

Conversely,

$((\bar{u},S_u),(\bar{v},S_v)) \in \overline{\rho_o}(A)$

$\Rightarrow S_u = S_v$ and $(\exists u' \equiv u, v' \equiv v)((u',v') \in \rho(A))$,

$\Rightarrow (u,v) \in \rho(R_1;A;R_2)$ for some sequences R_1, R_2 of sets and resets,

$\Rightarrow (u,v) \in \rho(R_1;R_2;A)$ by B4,

$\Rightarrow \exists w(u \equiv w \wedge (w,v) \in \rho(A))$,

$\Rightarrow u \equiv w \wedge S_w = S_v \wedge (w,v) \in \rho(A)$ by B9,

$\Rightarrow (u,v) \in \rho(A)$ by $S_u = S_v$ and lemma 2.3.

$\underline{x\dagger}$. $(u,v) \in \rho(x\dagger)$

$\Rightarrow u \equiv v \wedge S_v = S_u \cup \{x\}$ by B9, B1(c),

$\Rightarrow ((\bar{u}, S_u),(\bar{v}, S_v) \in \bar{\rho}(x\dagger)$.

Conversely,

$((\bar{u},S_u),(\bar{v},S_v)) \in \bar{\rho}(x\dagger)$

$\Rightarrow S_v = S_u \cup \{x\} \wedge \bar{u} = \bar{v}$.

By B1(a) there is a w such that $(u,w) \in \rho(x\dagger)$,

$\Rightarrow S_w = S_u \cup \{x\} = S_v \wedge \bar{w} = \bar{u}$,

$\Rightarrow w = v$ by lemma 2.3,

$\Rightarrow (u,v) \in \rho(x\dagger)$.

$\underline{\alpha \cup \beta}$. Trivial.

$\underline{\alpha;\beta}$. ($\Rightarrow$) Trivial.

($\Leftarrow$)

$((\bar{u},S_u),(\bar{v},S_v)) \in \bar{\rho}(\alpha;\beta)$

$\Rightarrow \exists w,s(((\bar{u},S_u),(\bar{w},s)) \in \bar{\rho}(\alpha)$ and $((\bar{w},s),(\bar{v},S_v))$
$\in \bar{\rho}(\beta))$.

Choose $w' \equiv w$ such that $S_{w'} = s$. This is possible by B1(a-d).

$\exists w'((\bar{u},S_u),(\bar{w}',S_{w'})) \in \bar{\rho}(\alpha)$ and $(\bar{w}',S_{w'}),(\bar{v},S_v)$
$\in \bar{\rho}(\beta))$,

$\Rightarrow (u,w') \in \rho(\alpha)$ $(w', v) \in \rho(\beta)$ by induction,

$\Rightarrow (u,v) \in \rho(\alpha;\beta)$

$\underline{\alpha^*}$. $\rho(\alpha^*) = \bigcup_n \rho(\alpha)^n$. We show by subinduction on n that $(u,v) \in \rho(\alpha^n)$ iff $((\bar{u}, S_u), (\bar{v}, S_v)) \in \bar{\rho}(\alpha^n)$.

$\underline{n=0}$. $(u,v) \in \rho(\alpha^0) \Leftrightarrow u = v$,

$\Leftrightarrow ((\bar{u},S_u),(\bar{v},S_v)) \in \bar{\rho}(\alpha^0)$.

n=1. Direct from the main induction hypothesis, for α is shorter than α^*.

n>1. $(u,v) \in \rho(\alpha^n)$

$\Rightarrow \exists w((u,w) \in \rho(\alpha)$ and $(w,v) \in \rho(\alpha^{n-1}))$,

$\Rightarrow \exists w(((u,S_u),(w,S_w)) \in \rho(\alpha)$ and $((w,S_w),$
$(v,S_v)) \in \rho(\alpha^{n-1})$ by the subinduction hypothesis,

$\Rightarrow ((u,S_u),(v,S_v)) \in \rho(\alpha^n)$.

Conversely,

$((\bar{u},S_u),(\bar{v},S_v)) \in \rho(\alpha^n)$

$\Rightarrow (\exists w,s)(((\bar{u},S_u),(\bar{w},s)) \in \rho(\alpha)$ and
$((\bar{w},s)(\bar{v},S_v)) \in \rho(\alpha^{n-1}))$.

Choose $w' \equiv w$ such that $S_{w'} = s$. Then

$\exists w'(((\bar{u},S_u),(\bar{w}',S_{w'})) \in \rho(\alpha)$ and

$((\bar{w}',S_{w'}),(\bar{v}, S_v)) \in \rho(\alpha^{n-1})$

$\Rightarrow \exists w'((u,w') \in \rho(\alpha)$ and $(w',v) \in \rho(\alpha^{n-1}))$
by the subinduction hypothesis,

$\Rightarrow (u,v) \in \rho(\alpha^n)$.

p?. $(u,v) \in \rho(p?)$

$\Leftrightarrow u = v$ and $u \models p$,

$\Leftrightarrow$ $u = v$ and $\bar{u}, S_u \models p$ by induction,

$\Rightarrow$ $((\bar{u}, S_u), (\bar{v}, S_v)) \in \bar{\rho}(p?)$

Conversely,

$((\bar{u}, S_u), (\bar{v}, S_v) \in \bar{\rho}(p?)$

$\Rightarrow$ $\bar{u} = \bar{v}$ and $S_u = S_v$ and $\bar{u}, S_u \models p$,

$\Rightarrow$ $u = v$ and $\bar{u}, S_u \models p$ by lemma 2.3,

$\Rightarrow$ $u = v$ and $u \models p$ by induction

$\Rightarrow$ $(u,v) \in \rho(p?)$. ∎

2.4. An upper bound on the complexity of B-PDL

By virtue of theorem 2.1 we already have a method of deciding satisfiability of B-PDL formulas: translate to PDL, and apply Fishcer and Ladner's [FL79] decision procedure for PDL. The resulting procedure requires nondeterministic triple exponential time in the worst case. We show here that we can do better by one exponential. In the next section we prove a deterministic double exponential time lower bound for B-PDL, indicating that further improvement of the upper bound is limited to making it deterministic instead of nondeterministic.

We extend Fischer and Ladner's proof of the decidability of PDL to Boolean PDL. By theorem 2.2, our decision method also works for B-PDL. A direct proof is also possible, following very closely the proof for PDL.

In outline, Fischer and Ladner's proof goes as follows: Given a model A satisfying P, we define a new model $\bar{A}$ whose states

are equivalence classes of states of A under a certain equivalence relation. $\bar{A}$ is shown to have a bounded number of states, and to satisfy p. A decision procedure for PDL is to guess a model of bounded size, and to test whether it satisfies p.

To extend the method to Boolean PDL, we must only show that $\bar{A}$ is Boolean, provided A is Boolean. In order to make $\bar{A}$ Boolean, we must strengthen the equivalence relation used by Fischer and Ladner. In so doing, we create more equivalence classes, and so increase the time required to decide p.

Theorem 2.4. Let p have length n and contain m distinct Boolean variables. Given any structure $A = (U,\Phi_o,\Sigma_o,\phi_o,\rho_o)$ satisfying p at some state u, and which is Boolean w.r.t. to variable set $V \subseteq \Phi_o$, there is a structure $\bar{A} = (\bar{U},\Phi_o,\Sigma_o,\overline{\phi_o},\overline{\rho_o})$ which satisfies p at state $\bar{u}$, and which is Boolean w.r.t. to V, and which has at most c^{n3^m} states for some constant c.

Proof. Following Fischer and Ladner, we define the closure cl(p) of a formula p to be the smallest set satisfying the following:

1. $p \in cl(p)$.
2. $p \vee q \in cl(p) \Rightarrow p,q \in cl(p)$.
3. $\neg p \in cl(p) \Rightarrow p \in cl(p)$.

4. $\langle A\rangle p \in cl(p) \Rightarrow p \in cl(p)$ for $A \in \Sigma_0$.

5. $\langle q?\rangle p \in cl(p) \Rightarrow q, p \in cl(p)$.

6. $\langle \alpha \cup \beta\rangle p \in cl(p) \Rightarrow \langle\alpha\rangle p, \langle\beta\rangle p, p \in cl(p)$.

7. $\langle \alpha;\beta\rangle p \in cl(p) \Rightarrow \langle\alpha\rangle\langle\beta\rangle p, \langle\beta\rangle p \in cl(p)$.

8. $\langle \alpha^*\rangle p \in cl(p) \Rightarrow \langle\alpha\rangle\langle\alpha^*\rangle p, p \in cl(p)$.

Fischer and Ladner show that if p has length n, then cl(p) has at most n members. Their equivalence relation over U is defined by $u \equiv_1 v$ iff $(\forall q \in cl(p))(u \models q$ iff $v \models q)$. We strengthen that equivalence relation to

$$u \equiv_2 v \text{ iff } (\forall q \in ecl(p))(u \models q \text{ iff } v \models q)$$

where ecl(p), the extended closure of p, is defined as follows:

Let $D = \{d_1; \ldots; d_m: d_i$ is either $x_i\uparrow$ or $x_i\downarrow$ or is missing, for $i=1, \ldots, m\}$. D has 3^m members.

$$ecl(p) = \{\langle d\rangle q: q \in cl(p), d \in D\}.$$

By definition, $\langle\lambda\rangle p$ is p.

Define $\bar{A} = (\bar{U}, \Phi_0, \Sigma_0, \bar{\phi}_0, \bar{\rho}_0)$ as follows:

$\bar{u} = \{v: u \equiv_2 v\}$;

$\bar{U} = \{\bar{u}: u \in U\}$;

$\bar{\phi}_0(P) = \{\bar{u}: u \models P\}$;

$\bar{\rho}_0(A) = \{(\bar{u}, \bar{v}): (u,v) \in \rho_0(A)\}$.

By the fact that ecl(p) has at most $n3^m$ members, we see that $\bar{U}$ has at most 2^{n3^m} members.

Lemma 2.5. For all $q \in ecl(p)$, $A,u \models q$ iff $\bar{A},\bar{u} \models q$.

Proof. Fischer and Ladner prove lemma 2.5 for all $q \in cl(p)$ based on the weaker relation $\equiv_1$. Their proof works for any stronger equivalence relation. Lemma 2.5 is extended to ecl(p) as follows:

$$u \models \langle d_1; \ldots; d_k \rangle q$$
$$\Rightarrow (\exists v_1, \ldots, v_k)((u,v_1) \in \rho(d_1) \wedge \ldots \wedge (v_{k-1},v_k) \in \rho(d_k) \wedge v_k \models q),$$
$$\Rightarrow (\exists \bar{v}_1, \ldots, \bar{v}_k)((\bar{u},\bar{v}_1) \in \bar{\rho}(d_1) \wedge \ldots \wedge (v_{k-1},v_k) \in \bar{\rho}(d_k) \wedge \bar{v}_k \models q)$$
by lemma 2.5 for cl(p),
$$\Rightarrow \bar{u} \models \langle d_1; \ldots; d_k \rangle q$$

Conversely, suppose $\bar{u} \models \langle d_1; \ldots; d_k \rangle q$. Then there must be a chain $u \equiv v_1 \rightarrow w_1 \equiv v_2 \rightarrow \ldots \rightarrow w_{k-1} \equiv v_k \rightarrow w_k$, where arrows mean $(v_i,w_i) \in \rho(d_i)$, and $\bar{w}_k \models q$. By lemma 2.5 for cl(p), $w_k \models q$. Hence $v_k \models \langle d_k \rangle q$. Since $\langle d_k \rangle q$ is in ecl(p), and $w_{k-1} \equiv v_k$, $w_{k-1} \models \langle d_k \rangle q$. Repeating that reasoning, we see that $u \models \langle d_1; \ldots; d_k \rangle q$. ∎

All that is left is to show that $\bar{A}$ obeys B1-B4. We may assume without loss of generality that every member of Φ_o appears in p. Then cl(p) contains every member of Φ_o. Let $\equiv$ be $\equiv_2$.

B1.

a) For every $\bar{u}$,

$\exists v((u,v) \in \rho(x\uparrow))$ by B1(a) in A,

$\Rightarrow (\bar{u},\bar{v}) \in \bar{\rho}(x\uparrow)$.

b) Dual to (a).

c) $(\bar{u},\bar{v}) \in \rho(x\uparrow)$

$\Rightarrow (\exists u' \equiv u, v' \equiv v)((u',v') \in \rho(x\uparrow))$

$\Rightarrow \bar{v}' = \bar{v}$ and $v' \vDash x$ by B1(c) in A,

$\Rightarrow \bar{v} \vDash x$ by lemma 2.5.

d) Dual to (c).

e) $(\bar{u},\bar{v}) \in \bar{\rho}(x\uparrow)$ and $\bar{u} \vDash P$

$\Rightarrow (\exists u' \equiv u, v' \equiv v)((u',v') \in \rho(x\uparrow)$

and $u' \vDash P)$ by lemma 2.5,

$\Rightarrow \bar{v}' = \bar{v}$ and $v' \vDash P$ by B1(e) in A,

$\Rightarrow \bar{v} \models P$ by lemma 2.5.

f) Dual to (e).

B2. We verify part (a) only.

$\bar{u} \models x$ and $(\bar{u},\bar{v}) \in \bar{\rho}(x\uparrow)$

$\Rightarrow$ $(\exists\, u' \equiv u, v' \equiv v)((u',v') \in \rho(x\uparrow)$ and $u' \models x)$ by lemma 2.5,

$\Rightarrow$ $u' = v'$ by B2 in A,

$\Rightarrow$ $\bar{u} = \bar{u}' = \bar{v}' = \bar{v}$.

B3. We verify part (a) only. Consider a state $\bar{u}$. We show that $(\bar{u},\bar{v}) \in \rho(x\downarrow)$ iff $(\bar{u},\bar{v}) \in \rho(x\uparrow;x\downarrow)$. There are two cases.

Case 1. $(\bar{u} \models x)$.

$\bar{u} \models x$ and $(\bar{u},\bar{v}) \in \bar{\rho}(x\downarrow)$

$\Rightarrow (\bar{u},\bar{u}) \in \bar{\rho}(x\uparrow)$ and $(\bar{u},\bar{v}) \in \bar{\rho}(x\downarrow)$ by B5 in $\bar{A}$,

$\Rightarrow (\bar{u},\bar{v}) \in \bar{\rho}(x\uparrow;x\downarrow)$.

Conversely,

$\bar{u} \models x$ and $(\bar{u},\bar{v}) \in \bar{\rho}(x\uparrow;x\downarrow)$

$\Rightarrow \exists\, \bar{w}((\bar{u},\bar{w}) \in \bar{\rho}(x\uparrow)$ and $(\bar{w},\bar{v}) \in \bar{\rho}(x\downarrow))$,

$\Rightarrow$ $\bar{w} = \bar{u}$ by B2 in $\bar{A}$,

$\Rightarrow$ $(\bar{u},\bar{v}) \in \bar{\rho}(x\downarrow)$.

Case 2. $(\bar{u} \models \sim x)$

$$(\bar{u},\bar{v}) \in \bar{\rho}(x\downarrow) \wedge \bar{u} \models \sim x$$

$\Rightarrow$ $\bar{u} = \bar{v} \wedge \bar{u} \models \sim x$ by B2 in A,

$\Rightarrow$ $u \models \sim x$ by lemma 2.5,

$\Rightarrow$ $(u,u) \in \rho(x\downarrow)$ by B5 in A,

$\Rightarrow$ $(u,u) \in \rho(x\uparrow;x\downarrow)$ by B3 in A,

$\Rightarrow$ $\exists\, w((u,w) \in \rho(x\uparrow)$ and $(w,u) \in \rho(x\downarrow))$,

$\Rightarrow$ $\exists\, \bar{w}((\bar{u},\bar{w}) \in \bar{\rho}(x\uparrow)$ and $(\bar{w},\bar{u}) \in \bar{\rho}(x\downarrow))$,

$\Rightarrow$ $(\bar{u},\bar{u}) \in \bar{\rho}(x\uparrow;x\downarrow)$,

$\Rightarrow$ $(\bar{u},\bar{v}) \in \bar{\rho}(x\uparrow;x\downarrow)$. by $\bar{u} = \bar{v}$.

For the converse we need to prove a lemma.

Lemma 2.6. Let d be either $x\uparrow$ or $x\downarrow$. If $(\bar{u},\bar{v}) \in \bar{\rho}(d)$ and $(\bar{u},\bar{w}) \in \bar{\rho}(d)$ then $\bar{v} = \bar{w}$.

Lemma 2.6 is a statement of determinism of $x\uparrow$ and $x\downarrow$ in $\bar{A}$. We can't use B7 directly, since the proof of B7 used B3.

Proof. We show that, for every $q \in ecl(p)$, $\bar{v} \models q \Leftrightarrow \bar{u} \models \langle d\rangle q \Leftrightarrow \bar{w} \models q$, thus showing that $v \equiv w$. By symmetry we need only show $\bar{v} \models q \Leftrightarrow \bar{u} \models \langle d\rangle q$.

($\Rightarrow$) $\bar{v} \models q \Rightarrow \bar{u} \models \langle d\rangle q$ because $(\bar{u},\bar{v}) \in \bar{\rho}(d)$.

($\Leftarrow$) Suppose $\bar{u} \models \langle d\rangle q$. For any q in ecl(p) it is easy to show that a formula which is equivalent to $\langle d\rangle q$ in all Boolean models is also in ecl(p). By that fact and lemma 2.5 we have

$u \models \langle d\rangle q$,

$\Rightarrow u \models [d]q$ by determinism of d in A,

$\Rightarrow \forall v'((u,v') \in \rho(d) \Rightarrow v' \models q)$,

$\Rightarrow \forall v'((u,v') \in \rho(d) \Rightarrow \bar{v}' \models q)$ by lemma 2.5.

But there is a $v' \equiv v$ such that $(u,v') \in \rho(d)$, so $\bar{v}=\bar{v}' \models q$. ∎

To continue the proof of B3, suppose $(\bar{u},\bar{v}) \in \bar{\rho}(x\uparrow;x\downarrow)$ and $\bar{u} \models \sim x$. Then there must be a $\bar{w}$ such that $(\bar{u},\bar{w}) \in \bar{\rho}(x\uparrow)$ and $(\bar{w},\bar{v}) \in \bar{\rho}(x\downarrow)$. By the definition of $\bar{\rho}$, there must be $u_1 \equiv u$, $w_1 \equiv w$, $w_2 \equiv w$ and $v_2 \equiv v$ such that $(u_1,w_1) \in \rho(x\uparrow)$ and $(w_2,v_2) \in \rho(x\downarrow)$. By B6 in A there is a $w_3 \neq u$ such that

$(u,w_3) \in \rho(x\uparrow)$ and $(w_3,u) \in \rho(x\downarrow)$. Similarly there is a v_3 such that $(w_3,v_3) \in \rho(x\downarrow)$. Taking each known member of $\rho(x\uparrow)$ and $\rho(x\downarrow)$ into its bar gives

$$(\bar{u},\bar{w}_3) \in \rho(x\uparrow),$$
$$(\bar{u}_1,\bar{w}_1) \in \rho(x\uparrow),$$
$$(\bar{w}_2,\bar{v}_2) \in \rho(x\downarrow),$$
$$(\bar{w}_3,\bar{u}) \in \rho(x\downarrow),$$
$$(\bar{w}_3,\bar{v}_3) \in \rho(x\downarrow).$$

By lemma 2.6 and the fact that $\bar{u} = \overline{u_1}$, we get $\bar{w}_1 = \bar{w}_3$. By another application of lemma 2.6, using $\bar{w} = \bar{w}_1 = \bar{w}_2 = \bar{w}_3$, we get $\bar{v} = \bar{v}_2 = \bar{v}_3 = \bar{u}$. By B5 in A, $(\bar{u},\bar{v}) \in \rho(x\downarrow)$.

B4. We verify (a) only. Suppose $(\bar{u},\bar{v}) \in \rho(A;x\uparrow)$. Then there must be $u' \equiv u$, w', $w'' \equiv w'$ and $v'' \equiv v$ such that

$$(u',w') \in \rho(A),$$
$$(w'',v'') \in \rho(x\uparrow).$$

By B1(a) in A, we can find v' such that $(w',v') \in \rho(x\uparrow)$. By determinism of $x\uparrow$ in $\bar{A}$, $\bar{v}' = \bar{v}'' = \bar{v}$.

$$(u',w') \in \rho(A) \text{ and } (w',v') \in \rho(x\uparrow)$$
$$\Rightarrow (u',v') \in \rho(A;x\uparrow),$$
$$\Rightarrow (u',v') \in \rho(x\uparrow;A) \qquad \text{by B4 in A,}$$
$$\Rightarrow \exists z((u',z) \in \rho(x\uparrow) \text{ and } (z,v') \in \rho(A)),$$
$$\Rightarrow \exists \bar{z}((\bar{u}',\bar{z}) \in \bar{\rho}(x\uparrow) \text{ and } (\bar{z},\bar{v}') \in \bar{\rho}(A)),$$
$$\Rightarrow (\bar{u}',\bar{v}'') \in \bar{\rho}(x\uparrow;A),$$

$\Rightarrow (\bar{u},\bar{v}) \in \bar{\rho}(x\uparrow;A)$.

The converse is proved in a similar manner. Its proof is omitted.

This concludes the proof that $\bar{A}$ is Boolean, and the proof of theorem 2.4. ∎

Theorem 2.7. There is an algorithm which recognizes SAT(B-PDL) and which runs in time at most c^{n3^m} on an input of length n containing n distinct Boolean variables, for some constant c.

Proof. A decision procedure for B-PDL can guess a structure of size at most d^{n3^m}, where d is the constant of theorem 2.4. It is left to the reader to verify that it is possible to test that the structure is Boolean and that it satisfies p in time polynomial in the number of states in the structure. The running time of this algorithm is d^{kn3^m} for some k. Let $c = d^k$. ∎

The procedure just presented has two serious shortcomings. For one thing, it is nondeterministic. A deterministic procedure based on it would have a longer running time by an exponential. For another, it takes the worst case time on all formulas. Pratt [Pr78] presents a tableau method for PDL which is deterministic and which takes far less than the worst case time on some inputs. The tableau method constructs a model for p, as our method does, but

instead of blindly searching for a model, the tableau method uses p to guide the construction of a model for p. It appears that conditions B1-B4 can be enforced on the model without affecting the rest of the construction procedure. When Pratt's method calls for the creation of a new state, the extension to Boolean PDL creates 2^m new states, associating a different subset of Boolean variables with each. We do not go into detail here on the extension of Pratt's tableau method for Boolean PDL, or attempt to prove the method correct.

2.5 A lower bound for B-PDL

This section is devoted to proving that SAT(B-PDL) is not solvable in deterministic time c^{n2^m} for some constant $c > 1$. The proof follows that of Fischer and Ladner for PDL. An outline of the method of proof is as follows: We show that B-PDL formulas can efficiently simulate computations of an $n2^m$ space bounded alternating Turing Machine, thus proving that SAT(B-PDL)is at least as difficult as the acceptance problem for such machines. By results of Chandra and Stockmeyer [CS76] and Kozen [Ko76] we can translate an alternating space bound into a deterministic time bound one exponential larger. In order to complete the proof, we need a result of abstract complexity theory which amounts to a compression theorem for functions

of several variables. As we are not aware of such a theorem in the literature, we prove it here.

For completeness, we give a definition of an alternating Turing Machine, taken from [FL79]. A one-tape ATM is a seven-tuple $M = (Q,\Delta,\Gamma,b,\delta,q_o,U)$ where

Q is a set of states,

Δ is the input alphabet,

Γ is the tape alphabet,

$b \in \Gamma - \Delta$ is the blank symbol,

$\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{L,R\})$ is the next move relation,

$U \subseteq Q$ is the set of universal states,

$E = Q - U$ is the set of existential states.

A <u>configuration</u> is a member of $\Gamma^* Q \Gamma^+$. A <u>universal configuration</u> is a member of $\Gamma^* U \Gamma^+$, and an <u>existential configuration</u> is a member of $\Gamma^* E \Gamma^+$. $\beta = x'q'\sigma'y'$ is a <u>next configuration</u> of $\alpha = x q \sigma y$ if for some $\tau \in \Gamma$, either

1) $(q,\sigma,q',\tau,L) \in \delta$ and $x'\sigma' = x$ and $y' = \tau y$,

or

2) $(q,\sigma,q',\tau,R) \in \delta$ and $x' = x\tau$ and $\sigma'y' = y$ or $(y = y' = \lambda$ and $\sigma' = b)$.

A <u>computation sequence</u> is a sequence $\alpha_1, \ldots, \alpha_k$ of configurations, where α_{i+1} is a next configuration of α_i for $1 \leq i \leq k$. A <u>trace</u> of M is a set C of pairs (α,t), where α is a configuration and $t \in \mathbb{N}$, such that

1) if $(\alpha,t) \in C$ and α is a universal configuration, then for every next configuration β of α, there is a $t'<t$ for which $(\beta,t') \in C$;

2) if $(\alpha,t) \in C$ and α is an existential configuration, then there is some next configuration β of α and $t'<t$ for which $(\beta,t') \in C$.

The set accepted by M is

$L(M) = \{x \in \Delta^*:$ there is a trace C of M and a $t \in \mathbb{N}$ such that $(q_o x,t) \in C\}$.

Machine M <u>accepts x in space s</u> if there is a trace of M containing $q_o x$, each of whose configurations uses at most s tape cells.

<u>Definition</u>. (Fischer and Ladner) A <u>simplified trace</u> is a set of configurations which is equal to the set of first components of some trace.

<u>Lemma 2.9</u>. (Fischer and Ladner). If M never repeats a configuration, then $L(M) = \{x \in \Delta^*:$ there is a simplified trace of M which contains $q_o s\}$. ∎

We now show that B-PDL can efficiently simulate space bounded alternating Turing machines. Let $\langle n,m \rangle$ be a standard encoding of the pair (n,m) in alphabet Δ.

<u>Lemma 2.10</u>. Let $K \subseteq \Delta^*$ be accepted by an alternating Turing machine M which accepts every $\langle n,m \rangle \in K$ in space $n2^m$. There is a mapping f from Δ^* into B-PDL formulas such

that for every pair $\langle n,m \rangle$,

i) $\langle n,m \rangle \in K$ iff $f(\langle n,m \rangle)$ is satisfiable,

ii) $f(\langle n,m \rangle)$ has length $O(n+m)$ and contains $O(m)$ distinct Boolean variables,

iii) $f(\langle n,m \rangle)$ is computable in time polynomial in $n+m$.

Proof. We may assume without loss of generality that M never repeats a configuration on any computation sequence; for there must be some j such that j^{n2^m} bounds the number of configurations of M. We can construct a new machine M′ which on input $\langle n,m \rangle$ maintains a count on a new track, in j-ary, of the number of moves which M has made. M′ accepts $\langle n,m \rangle$ in space $n2^m$ iff M does so. By lemma 2.8, we need only consider simplified traces of M′.

A PDL structure represents an $n2^m$ space bounded simplified trace as follows: A configuration is represented as a chain of $m2^m$ states, linked by basic program A. Each state holds $\lceil n/m \rceil$ tape cells. The i^{th} state in a chain satisfies basic formula P_σ^j or H_j, respectively, if tape cell $i\lceil n/m \rceil + j$ contains σ or the head is reading cell $i\lceil n/m \rceil + j$, respectively for $\sigma \in \Gamma$, $j = 0, \ldots, \lceil n/m \rceil - 1$, $i = 0, \ldots, m2^m - 1$. Formula Q_q, for $q \in Q$, holds at the first state of the chain if the associated configuration is in state q. The "next move" relation between configurations is represented by basic program N, which

operates at the first state of a chain.

Before defining f(<n,m>), we define some abbreviations. Let $y_1, \ldots, y_k$, $k = \lceil \log(m2^m) \rceil$, be distinct Boolean variables. $y_1, \ldots, y_k$ represent an integer y in the range $[0, m2^m-1]$.

1. The following programs can be simulated by length O(m) programs:

a) $y = 0$?

b) $y \leftarrow y-1 \bmod m2^m$.

2. α^y can be simulated by a program of length $O(m + \ell(\alpha))$.

3. $y \leftarrow \text{random} = (y_1\uparrow \cup y_1\downarrow); \ldots; (y_k\uparrow \cup y_k\downarrow)$.

Bounded quantification is simulated by

$\forall y = [y \leftarrow \text{random}]$,

$\exists y = \langle y \leftarrow \text{random} \rangle$.

Formulas g1-g7 force a structure to represent a simplified trace.

g1. Every tape cell is present.

$[A^*]\langle A \rangle \text{true}$

g2. There is exactly one state.

$$\bigvee_{q \in Q} (Q_q \wedge \bigwedge_{q' \neq q} \neg Q_{q'})$$

g3. There is exactly one character per cell, and it is well defined.

$$[A^*] \bigwedge_{j=0}^{\lceil n/m \rceil - 1} \bigvee_{\sigma \in \Gamma} (P_\sigma \wedge \bigwedge_{\sigma' \neq \sigma} \sim P_{\sigma'}) \wedge \forall y \bigwedge_{j=0}^{\lceil n/m \rceil - 1} \bigwedge_{\sigma \in \Gamma}$$

$$([A^y]P_\sigma \vee [A^y]\sim P_\sigma).$$

g4. There is exactly one head position, and it is well defined.

$$\langle A^* \rangle \bigvee_{j=0}^{\lceil n/m \rceil - 1} (H_j \wedge \bigwedge_{i \neq j} \sim H_i) \wedge [A^*] (\bigvee_{j=0}^{\lceil n/m \rceil - 1} H_j \supset [A;A^*]$$

$$\bigwedge_{j=0}^{\lceil n/m \rceil - 1} \sim H_j) \wedge \forall y \bigwedge_{j=0}^{\lceil n/m \rceil - 1} ([A^y]H_j \vee [A^y]\sim H_j).$$

g5. The universal states behave correctly. Let

$$\text{MOVER} = \begin{cases} H_{j+1} & \text{if } j < \lceil n/m \rceil - 1, \\ y \neq m2^m - 1 \wedge \langle A^{y+1} \rangle H_0 & \text{if } j = \lceil n/m \rceil - 1, \end{cases}$$

$$\text{MOVEL} = \begin{cases} H_{j-1} & \text{if } j \neq 0, \\ y \neq 0 \wedge \langle A^{y-1} \rangle H_{\lceil n/m \rceil - 1} & \text{if } j = 0. \end{cases}$$

$$\forall y \bigwedge_{j=0}^{\lceil n/m \rceil - 1} \bigwedge_{\sigma \in \Gamma} \bigwedge_{q \in U} (Q_q \wedge \langle A^y \rangle (P_\sigma^j \wedge H_j) \supset$$

$$(\bigwedge_{q',\sigma'} \langle N \rangle (\text{MOVER} \wedge Q_{q'} \wedge \langle A^y \rangle P_{\sigma'}^j)$$

$$(q,\sigma,q',\sigma',R) \in \delta$$

$$\wedge \bigwedge_{q',\sigma'} \quad <N>(\text{MOVEL} \wedge Q_{q'} \wedge <A^y>P^j_{\sigma'})).$$

$(q,\sigma,q',\sigma',L) \in \delta$

(Empty conjunctions are defined to be true.)

g7. The existential states behave correctly. Similar to g6.

Let $<n,m> = \sigma_1 \ \cdots \ \sigma_k$. The initial configuration q_o $<n,m>$ is enforced by

$$h = Q_{q_o} \wedge (\forall y < mk/n) \bigwedge_{j=o}^{\lceil n/m \rceil - 1} <A^y>P^j_{\sigma \lceil n/m \rceil y+j}.$$

Finally, define

$$f(\langle n,m\rangle) = h \wedge [N^*](g_1 \wedge \ldots \wedge g_7).$$

$f(\langle n,m\rangle)$ satisfies conditions (ii) and (iii) by inspection. Given a simplified trace of M on input $\langle n,m\rangle$ using space $n2^m$ it should be clear how to construct a model which satisfies $f(\langle n,m\rangle)$. Conversely, given $A,u_o \models f(\langle n,m\rangle)$ we can find a simplified trace for M on x. The g formulas are sufficient to ensure that there is a configuration associated with each state accessible from u_o by N^*. The g conditions also ensure that, at least for some subset U_o of the states U of A, the set of configurations associated with members of U_o form a simplified trace of M which accepts $\langle n,m\rangle$. Details of the proof are omitted. ∎

A compression theorem

Theorem 2.12 is the compression theorem which we require to finish the lower bound proof.

Definition. $(x_1, \ldots, x_n) \leq (y_1, \ldots, y_n)$ iff

$x_1 \leq y_1 \wedge \ldots \wedge x_n \leq y_n$.

Lemma 2.11. Let $S \subseteq \mathbb{N}^n$, and suppose that no two elements of S are comparable by $\leq$. Then S is finite.

Proof. We prove a stronger form.

Claim. Let $0 \leq k \leq n$, and let $S \subseteq \mathbb{N}^n$ be such that no two elements are comparable, but all elements are compar-

able in their first k positions (i.e., if $(u_1, \ldots, u_k, \ldots, u_n)$ and $(u_1', \ldots, u_k', \ldots, u_n')$ are both in S, then either $(u_1, \ldots, u_k) \leq (u_1', \ldots, u_k')$ or $(u_1', \ldots, u_k') \leq (u_1, \ldots, u_k)$). Then S is finite.

Proof of claim. By induction on n-k.

Let $x = (x_1, \ldots, x_k, \ldots, x_n) \in S$ be chosen with minimal $(x_1, \ldots, x_k)$. Such an element exists by the total order assumption on the first k positions. For every $u = (u_1, \ldots, u_n) \in S$ not equal to x, there must be an $i, k<i\leq n$, for which $u_i<x_i$, for otherwise u and x would be comparable. We count the members of S with $u_i = v$ separately for each $v<x_i$. We may assume without loss of generality that $i = k+1$, otherwise reordering the components. Let

$$S_v = \{(u_1, \ldots, u_n) \in S: \ u_i = v\}.$$

The first k+1 positions of S_v are totally ordered, and no two elements of S_v are comparable. Hence S_v is finite by induction. Finally, $|S| \leq \sum_{i=k+1}^{n} \sum_{v=1}^{x_i} |S_v|$, which is finite. ∎

Theorem 2.12. (Fischer) Let $t(n_1, \ldots, n_k) \geq n_1 + \ldots + n_k$ be a recursive, honest function (computable in time polynomial in t). There exists a set X such that for every deterministic Turing machine M accepting X, M runs for time at least $t(n_1, \ldots, n_k)$ on input $\langle n_1, \ldots, n_k \rangle$ for all

but finitely many values of $n_1, \ldots, n_k$. Moreover, there is a deterministic machine M_o which accepts X, and which takes at most $cn_1 \ldots n_k(n_1 + \ldots + n_k)t(n_1, \ldots, n_k)^{c'}$ time on input $\langle n_1, \ldots, n_k \rangle$ for some constants c and c'.

Proof. For clarity we prove theorem 2.12 for functions of two variables. The extension to k variables is straightforward. We use a priority argument. Let the deterministic machines be ordered in the usual manner, and let L(e) be the set accepted by the e^{th} machine. We define X by describing machine M_o which accepts X. On an input which is not an ordered pair, M_o halts and does not accept. On input <n,m>, M_o runs stage (i,j) for all $(i,j) \leq (n,m)$ in an order consistent with the partial order $\leq$, starting with (0,0). Each stage produces a value and a cancellation list. <n,m> is accepted if stage (n,m) returns value 1.

Stage (n,m). Let $C = \bigcup_{(i,j) < (n,m)} C(i,j)$, where C(i,j) is the cancellation list of stage (i,j). Let t = t(n,m). Run each of the first n+m machines for at most t steps on input <n,m>. Let e be the first machine to halt. (If no such e exists, set C(n,m) = C and return 0). Let C(n,m) = C ∪ {e}, and return 1 if and only if e does not accept <n,m>.

Suppose $L(e) = X$. Then e is never cancelled. Every $e' < e$ is cancelled during stage (i,j) for only finitely many values of i and j. To see this, let

$$S_{e'} = \{(i,j): \ e' \text{ is cancelled during stage } (i,j)\}.$$

Clearly, if $(i,j) \in S_{e'}$, then $(n,m) \notin S_{e'}$ for any $(n,m) > (i,j)$, for e' will be in the set C computed at stage (n,m). Hence, the elements of $S_{e'}$ are pairwise incomparable, and $S_{e'}$ is finite by lemma 2.11. Let

$$q = \max(\{n+m: \ (n,m) \in \bigcup_{e'<e} S_{e'}\} \cup \{e\}).$$

For every (n,m) with $n+m>q$, it must be the case that machine e runs for more than t(n,m) steps on input <n,m>, otherwise e would have been cancelled at stage (n,m). Hence e runs for more than t(n,m) steps for all but the finitely many values of <n,m> for which $n+m \leq q$.

Machine M_o computing X runs in time at most $n \cdot m \cdot$(time per stage) on input <n,m>. There are at most n+m machines to simulate at each stage, and each can be simulated in time O(t log t). The time to compute t is $O(t^{c'})$ by the honesty of t. Putting this together gives the time bound for M_o. ∎

We require an extension of the result of Chandra and Stockmeyer [CS76] and Kozen [Ko76] relating alternating space to deterministic time. Their theorem states that

$ASPACE(s(n)) = \bigcup_{c>0} DTIME(c^{s(n)})$ for any suitably honest s. The proof relies on a simulation of each type of machine by the other, and is easily extended to several variables.

Theorem 2.13. Let $s(n_1, \ldots, n_k)$ be constructable. Given any alternating Turing machine M which runs in space $s(n_1, \ldots, n_k)$ on input $\langle n_1, \ldots, n_k \rangle$, there is a deterministic Turing machine M accepting L(M) which runs in time $c^{s(n_1, \ldots, n_k)}$ for some constant c. Conversely, given deterministic machine M running in time $c^{s(n_1, \ldots, n_k)}$ on input $\langle n_1, \ldots, n_k \rangle$, there is an alternating Turing machine M accepting L(M') which runs in space $s(n_1, \ldots, n_k)$ on input $\langle n_1, \ldots, n_k \rangle$.

We are ready to prove the lower bound for B-PDL.

Theorem 2.14. (Fischer). Let M be any machine accepting SAT (B-PDL). Then there are constants d and d' such that for all but finitely many values of n and m there is a formula $F_{n,m}$ of length at most (n+m) containing at most m distinct Boolean variables, on which M runs for more than $2^{dn2^{d'm}}$ steps.

Proof. Let $t(n,m) = 2^{an^{2m}}$ and $t_2(n,m) = cnm(n+m)t(n,m)^{c'}$, where c and c' are the constants of theorem 2.12. There is a constant b such that $t_2(n,m) < 2^{bn2^m}$. Let X be the set of theorem 2.12. By theorem 2.13, there is an alternating machine A accepting X which runs in space $n2^m$ on input $\langle n,m \rangle$. Lemma 2.10 asserts the existence of a formula $G_{n,m}$ of length at most $c_1(n+m)$ with at most

$c_2 m$ Boolean variables such that $G_{n,m}$ is satisfiable iff $\langle n,m\rangle \in X$. Moreover, $G_{n,m}$ can be found in time $(n+m)^k$. Hence, the following is a procedure for accepting X.

1. Given $\langle n,m\rangle$, construct $G_{n,m}$.
2. Test if $G_{n,m}$ is satisfiable by running M. If so, accept $\langle n,m\rangle$, else reject $\langle n,m\rangle$.

Let $T(n,m)$ be the time M spends to decide $G_{n,m}$. Then the above procedure accepts X in time $(n+m)^k + T(n,m)$. By choice of X, $(n+m)^k + T(n,m) \geq t(n,m)$ for all but finitely many values of n and m. Hence there is a constant e such that $T(n,m) \geq 2^{en2^m}$. Let $c = \max(c_1,c_2)$, and let $F_{n,m} = G_{\lfloor \frac{n}{c} \rfloor, \lfloor \frac{m}{c} \rfloor}$. $F_{n,m}$ has length at most n+m and contains at most m Boolean variables. M decides $F_{n,m}$ in time $T(\lfloor \frac{n}{c} \rfloor, \lfloor \frac{m}{c} \rfloor) \geq 2^{e\lfloor \frac{n}{c} \rfloor 2^{\lfloor \frac{m}{c} \rfloor}} \geq 2^{dn2^{d'm}}$ for some d and d', for all sufficiently large n and m. ∎

2.6. Multiple variable complexity bounds

In proving a lower bound for B-PDL which has nearly the same form as our upper bound, both being functions of n and m, we have demonstrated both the desirability and feasibility of proving bounds which are functions of more than just the length of the input. For most problems, some inputs are easier than others. For some, such as

SAT(B-PDL), there are natural parameters of the input which appear in tight complexity bounds. Another example of such a problem is the not-everything problem for extended regular expressions [St75], which is decidable in time $2^{2^{\cdot^{\cdot^{\cdot^{2^{cn}}}}}}\Big\}\, m+1$ for expressions of length n with m complement symbols, as opposed to $2^{2^{\cdot^{\cdot^{\cdot^{2^{c_1 n}}}}}}\Big\}\, c_2 n$ when m is left unspecified.

In our compression theorem we consider only inputs which are ordered pairs, showing that, even when inputs are restricted to ordered pairs, there are arbitrarily hard problems. For proving lower bounds, that is enough, and it results in a fairly clean proof. In general, though, a complexity bound is a function t(x) of the input, which might have the form t(n(x)) or t(n(x),m(x)), where n and m are simple functions of the input, such as its length. There is a need for a theory of more general complexity bounds that the traditional ones which depend only on the length of the input.

2.7. Conclusion

By showing that B-PDL is decidable, we have shown that PDL with any or all of the following extensions is decidable, provided basic programs represent indivisible actions: concurrency, assignment and quantification over

bounded integers, gotos, labeled programs with formulas having access to labels, global invariance, "preserves," while and until (unnested). By the fact that B-PDL is no more expressive than PDL, we find that all of the above concepts can be simulated in PDL. We can view that fact two ways. One way is to view PDL as a surprisingly rich language. Another view is that any language which hopes to be more powerful than PDL must be able to express more than the above, or to deal with basic programs which are not indivisible.

One way to handle concurrency is by the brute force method of trying all possible interleavings. Owicki [OG76] presents a proof system for proving partial correctness which permits reasoning without considering all possible interleavings. The B-PDL simulation of concurrency also permits a more efficient way of handling concurrency than considering all possible interleavings. Improved efficiency results due to the exponential gap between our decision method for B-PDL and the naive method of translating from B-PDL into PDL, and then deciding the resulting PDL formula.

It would seem a reasonable criterion of any logic of concurrent programs that it be capable of dealing with concurrency with more finesse than can be achieved by reducing concurrent programs to <u>while</u> (or PDL) programs. For otherwise we might as well just use PDL to begin with. This observation applies equally well to decision

procedures and proof systems. Any axiom system for $PDL_{//}$ which ultimately relies on reducing away concurrency by expressing it in terms of $\cup$, ; and * (such systems have been shown to us more than once) is misguided.

It is an open problem to find a complete proof system for B-PDL. By the remarks above, an acceptable system would not rely on a costly elimination of Boolean variables. We have remarked that the axioms of condition B1 cannot form a complete axiomatization of B-PDL, when added to a system for PDL. Any system for B-PDL must somehow express the independence of $x\uparrow$ and other basic programs.

Chapter 3

A General Process Logic

In this chapter we describe a logic GPL in which variables and quantifiers are used to express properties of a given process π. By excluding programs from the syntax of GPL, we greatly simplify our analysis. Valid sentences of GPL are those which every process must obey, rather than those which some particular, potentially very complicated program must obey. It is possible to add programs to GPL, by adding new predicates.
GPL with programs is very similar to a version of Parikh's Second Order Process Logic (SOPL), in which first order quantifiers range over occurrences of states rather than over states. In contrast to standard SOPL, which is undecidable by Parikh [Pa78], we do not know whether GPL is decidable. However, we give two restrictions of GPL, each of which is decidable. The first is a semantic restriction, in which processes are required to be closed, in the sense that any path which can be followed arbitrarily long can also be followed infinitely long. In other words, processes must exhibit bounded nondeterminism. The second restriction of GPL is syntactic, and is shown in

Chapter 4 to be very nearly expressively equivalent to the modal logic MPL. The theories of GPL and both of its restrictions are nonelementary.

3.1. Introduction

Many statements which we wish to make about processes concern the order of events on paths. A simple example is global invariance: at every time instant t, P(t) holds. For another example, suppose that P(t) represents a message sent at time t, and Q(t) is an acknowledgement. We may require that 1) for every time instant t, if P(t) holds, then there is a later time t′ when Q(t′) holds, and 2) for every time instant t for which Q(t) holds, there is a previous time t′ when P(t′) holds. The predicate calculus of an order immediately volunteers itself as a process logic. The parts of such a process logic are as follows:

1. Variables are called stage variables. A stage, or a time, is a finite path, which gives the history of a computation. Because it is impossible for a computation to proceed beyond a block, stage variables must range only over legal sequences.

2. $s \leq t$, where s and t are stage variables, is a formula. In terms of paths, $\leq$ is simply the prefix relation.

3. P(t), where $P \in \Phi_0$ is a basic predicate,

is a formula. The truth value of P(t) depends only on the final state of t.

We generally want to make statements about the paths in some set π. For example, to state that P(t) is globally invariant over π, we would say that, for every path h in π, and every stage t on h, P(t) holds. GPL must have some means of quantifying h, and selecting t on h. There are two obvious methods which we could use.

1. h can be specified implicitly by the semantics, either by letting h be a part of the environment, or by implicitly universally quantifying h before every formula. These approaches are taken in [Pn77,Pn79,GPSS80].

2. We can introduce variables which range over paths, and write "t on h" explicity as $t \leq h$.

Below we show that the first approach is inadequate; hence we choose the second. Path variables and stage variables both range over paths, so we could get by with a single type of variable, along with some additional predicates such as legal (x). Harel et al. [HKP80] seem convinced that a single type of variable is better than two, and define a logic based on a single type of variable. However, path variables and stage variables really have different purposes. Natural restrictions are easily expressed in terms of the two different types of variable. Therefore we choose to define path variables separately from stage variables.

1. Path variables range over paths in π. In order to allow for the possibility of diverging or blocking, we must allow path variables to range over infinite and illegal paths as well as terminating paths.

2. $t \leq h$ is a formula, for t a stage variable and h a path variable. ($t \leq s$ is still allowed).

Informal specification of the logic GPL is almost complete. (A complete formal specification is given in section 3.2.) There is still one serious hole which needs filling. In the language given so far, while it is possible to state that path h can make no more progress at stage t, as $(t \leq h \wedge \forall s(s \leq h \supset s \leq t))$, it is impossible to distinguish a path which is blocked at stage t from one which is terminated at stage t. We introduce the formula H(t,h) which means path h is terminated (or halted) at stage t. In terms of paths, H is just the equality predicate. We prefer H to = for the reasons that the parameters are of different types, and that H(t,h) corresponds to the atomic formula H of MPL.

Why path variables?

The subset TL of GPL without path variables or blocked paths has been studied as a viable process logic by Pnueli and Gabbay, et al. [Pn77,GPSS80] who show it capable of expressing a number of significant properties of processes. However, there are some important properties of processes which appear to be expressible only by using path variables. Some, such as the absence of deadlock,

depend on the existence of blocked paths, which most other authors have not considered (see Pratt [Pr78] for an exception). Others are more basic.

1. The fundamental property of global invariance,

$$GI(p) = (\forall h \varepsilon \pi)(\forall t \underline{\leq} h)p(t)$$

depends at least on a single universally quantified path variable. The approach of letting the path be part of a model is not suitable to describing processes which are sets of paths. The alternative approach of implicitly prefixing every formula by $(\forall h \varepsilon \pi)$, and permitting no further quantification of path variables, results in a logic which is not closed under semantic negation, for the negation of GI(p) begins $(\exists h \varepsilon \pi)$ ... In such a system it is possible for both p and $\neg p$ to fail to hold in a given model. It is out of the question to attempt to disprove a property when we can't even state its negation. Furthermore, an algorithm for deciding satisfiability of a system which is not closed under semantic negation does not immediately extend to deciding validity the way it does for logics which are closed under negation.

2. There are really two different notions of the "future" at a given stage τ. One is the linear, determined future on a given path, or the future as it <u>will</u> happen. The other is the branching, undetermined future of all paths of which τ is a prefix, or the future as it <u>might</u> happen. Lamport [L80] shows that neither notion

of future is definable in terms of the other. Lamport argues that the linear notion of future is more appropriate for reasoning about concurrent processes, while the branching notion is more appropriate for reasoning about sequential processes (e.g., PDL uses branching futures). (We find neither completely adequate.) Since our system is to treat sequential and concurrent processes uniformly, we require both notions of future. "Throughout the future from time t" is expressed as $\forall s(t \underline{<} s \underline{<} h \supset \ldots)$ in the linear case, and $(\forall h \underline{>} t)\forall s(t \underline{\leq} s \underline{\leq} h \supset \ldots)$ in the branching case.

3. We mentioned in Chapter 1 that we would give the writer of formulas the power to be his own oracle, making choices when he sees fit. Path quantifiers are the mechanism for making new choices. The absence of deadlock statement, assuming (or simulating) an oracle which does its best to resolve blocks without backtracking, can be expressed in GPL by the following formula, with nested, alternating path quantifiers.

$$(\forall h\ )(\forall t \underline{\leq} h)(H(t,h) \vee (\exists h' \underline{\geq} t)(\exists t')(t < t' \underline{\leq} h')).$$

Relation of GPL to SOPL

Except for the absence of programs, GPL is very similar to Parikh's Second Order Process Logic (SOPL) [Pa78]. Both have two kinds of variables, and a means of ordering

occurrences of states on a path. The major difference is that in SOPL first order variables range over states, while their analogs in GPL range over stages. As a consequence of that difference, while in SOPL it is possible to express that some state occurs twice on a given path, the same is not true for GPL. Thus, regardless of programs, GPL cannot simulate SOPL. However, we know of no really useful statement which can be made in SOPL, but not in GPL (ignoring programs), and, since Parikh has shown that SOPL is undecidable, we may not want the full power of SOPL. We do not know whether GPL is decidable.

When Parikh defines the restriction SOAPL of SOPL, he changes the meaning of first order quantifiers, letting them range over stages rather than states. But he restricts the use of path quantifiers to such an extent that they can no longer be used as we have used them in our absence of deadlock statement. In SOAPL, every time a path is quantified, it is restarted, and bound to a new process. In Chapter 5 we show that, when programs are added to GPL, GPL is strictly more expressive than SOAPL.

The logics of Pratt [Pr78], Pnueli [Pn79] and Nishimura [N79] all restrict the use of path quantifiers the way SOAPL does, so they can't be used as they are in our absence of deadlock statement. Our less restrictive use of path quantifiers is a major difference between GPL (and MPL) and most process logics proposed to date.

3.2. Formal definition of GPL

The syntax and semantics of GPL are given below. The truth value of a GPL formula is determined by an environment $E = (A,f)$, consisting of a structure $A = (U,\pi,\Phi_0,\phi_0)$, which supplies a process π over the set U of states and interprets the basic predicates, and a binding f of variables to values, with $f(h) \in \pi$ for h a path variable, and $f(t) \in pre(\pi)$ for t a stage variable. $pre(\pi)$ is the set of all finite legal prefixes of members of π. Let $P \in \Phi_0$ be a basic predicate, p and q be GPL formulas, s and t be stage variables, and h be a path variable.

1. $P(t) \in GPL$; $E \models P(t)$ iff end $(f(t)) \in \phi_0(P)$.
2. $H(t,h) \in GPL$; $E \models H(t,h)$ iff $f(t) = f(h)$.
3. a) $\sim p \in GPL$; $E \models \sim p$ iff not $(E \models p)$;
   b) $(p \vee q) \in GPL$; $E \models p \vee q$ iff $E \models p$ or $E \models q$.

The usual Boolean operators true, false, $\wedge$, $\supset$, etc. are defined in terms of $\vee$ and $\sim$.

4. a) $(t \leq s) \in GPL$; $E \models t \leq s$ iff $f(t) \leq f(s)$;
   b) $(t \leq h) \in GPL$; $E \models t \leq h$ iff $f(t) \leq f(h)$.

The semantic $\leq$ is the prefix relation.

5. $\exists t\ p \in GPL$; $E \models \exists t\ p$ iff $(\exists \tau \in pre(\pi))(E_t^\tau \models p)$.
6. $\exists h\ p \in GPL$; $E \models \exists h\ p$ iff $(\exists \psi \in \pi))(E_h^\psi \models p)$.

$E_t^\tau(E_h^\psi)$ is the environment which assigns $f(t) = \tau (f(h) = \psi)$, with all other assignments being the same as in E. It is

well known that the relations $<$, $=$ and $t=succ(t')$ (successor) can be expressed in terms of $\leq$. For example,

$(t = succ(t'))$ = t is the next stage following t',

$\equiv t' < t \wedge \forall s(s \leq t' \vee t \leq s)$.

We are ready to prove some technical results about GPL. We begin by defining a nonstandard semantics of GPL. Nonstandard GPL has the advantage of being more closely related to some other logics than is standard GPL, though standard GPL more closely reflects our intuition about the nature of processes and predicates. Since we show that the satisfiable formulas are the same under either semantics, we can interchange the two freely.

3.3. Nonstandard GPL

In most versions of the predicate calculus, an uninterpreted predicate P(t) is interpreted freely over the same set as t ranges over. But in GPL, basic predicates apply to stage variables, while they depend for their truth value only on the final state of a stage. Thus it is required that $P((u,\lambda)) \equiv P((u,<u \to u>))$. A natural extension to GPL is to permit the truth value of predicates to depend on the whole stage, not just the final state. That extension is nonstandard GPL. The logic N-GPL is defined exactly as GPL, replacing $\models$ by $\models^N$, with the exceptions that in a nonstandard structure A^N =

(U,π,Φ_0,ϕ_0^N), $\phi_0:\Phi_0 \to P(S(U))$ is a more general interpretation of basic predicates, and rule (1) for GPL is replaced by

$$1^N.\quad P(t) \in \text{N-GPL};\quad E \models^N P(t) \quad \text{iff} \quad f(t) \in \phi_0^N(P).$$

A natural question is whether the satisfiable (or valid) formulas of GPL and N-GPL are the same. The answer is yes.

Before proving that, we make a short digression concerning a strengthening of GPL. Rather than letting path variables range over π, and stage variables range over $pre(\pi)$, we could let path variables range over all paths in $\Psi(U)$, and stage variables range over all stages in $S(U)$. The ranges of quantifiers can be explicitly bounded using the special predicate $h \in \pi$.

Though the stronger version of GPL is more expressive than GPL, it is not as well behaved. The standard and nonstandard semantics do not yield the same satisfiable formulas in the strong version of GPL, for we can write a formula which says that P holds for exactly one stage, as

$$Q \equiv \exists t \forall s(P(t) \wedge (P(s) \supset s = t)).$$

Q is certainly satisfiable under the nonstandard semantics, in either GPL or the strong version of GPL. But under the standard semantics, if $P(u,\sigma\langle v \to w\rangle)$ holds, then so

must $P(u,\sigma\langle v\to w\rangle\langle w\to w\rangle)$; hence Q is not satisfiable in the standard strong version of GPL.

Nevertheless, the satisfiable formulas of GPL are the same under the standard and nonstandard semantics. The reason is, intuitively, that by limiting the range of quantifiers, structures have more control over the truth of formulas. Q is satisfiable in standard GPL; simply let π be the singleton set $\{(u,\lambda)\}$.

Since the purpose of GPL is to describe the set π, it is unnatural to permit variables to range over a set larger than π and its prefixes. Therefore we study the better behaved logic GPL.

Theorem 3.1. SAT(GPL) = SAT(N-GPL)

Proof. The inclusion $SAT(GPL) \subseteq SAT(N\text{-}GPL)$ is trivial, for ϕ_o^N can assign the same truth value to all stages which end on the same state.

Suppose $E^N = ((U^N,\pi^N,\Phi_o,\phi_o^N),f^N)$ is a nonstandard environment, and $E^N \models p$. We construct a standard environment $E^S = (U^S,\pi^S,\Phi_o,\phi_o^S),f^S)$ for p as follows, letting each state on a path remember the entire history up to its position.

$U^S = S(U^N)$.

$K:\Psi(U^N) \to \Psi(U^S)$.

For finite Ψ,

$K((u,\lambda)) = (u,\lambda)$,

$K((u,\langle v\to w\rangle)) = (u,\langle(u,\lambda) \to (u,\langle v\to w\rangle)\rangle)$,

$$K(\psi\cdot\langle u\to v\rangle\langle w\to x\rangle) = K(\psi\cdot\langle u\to v\rangle) \cdot \langle \psi\cdot\langle u\to w\rangle \to \psi\cdot\langle u\to v\rangle\langle w\to x\rangle \rangle.$$

For infinite ψ, $K\psi$ is the limit of $K\tau$ for all $\tau<\psi$.

$$\pi^S = \{K\psi: \psi\in\pi^N\}$$
$$\phi_o^S = \phi_o^N.$$
$$f^S = K \circ f^N.$$

The states of E^S are the stages of E^N. K replaces each state u in ψ by the prefix of ψ up to u. For example,

$$(*)\quad K(u,\langle u\to v\rangle\langle w\to x\rangle) = (u,\langle (u,\lambda) \to (u,\langle u\to v\rangle) \rangle \langle (u,\langle u\to w\rangle) \to (u, \langle u\to v\rangle\langle w\to x\rangle)\rangle).$$

Notice that the second transition of the right hand side of (*) begins with $(u,\langle u\to w\rangle)$, not $(u,\langle u\to v\rangle)$. If w=v, then it makes no difference, and both sides of (*) are legal. On the other hand, if $w\neq v$, then both sides are illegal. It can be shown that

K1. ψ is legal iff $K\psi$ is legal.

Other easily proved facts about K are

K2. ψ = end $(K\psi)$ for all finite ψ;

K3. $\psi_1 \leq \psi_2$ iff $K\psi_1 \leq K\psi_2$.

Theorem 3.1 follows immediately from the following claim:

Claim. For every p, E^N and associated E^S,

$E^N \models^N p$ iff $E^S \models p$.

Proof. By induction on the length of p.

P(t). $E^N \models^N P(t) \Leftrightarrow f^N(t) \in \phi_0^N(P)$,

$\Leftrightarrow f^N(t) \in \phi_0^S(P)$,

$\Leftrightarrow$ end $(Kf^N(t)) \in \phi_0^S(P)$ by K2,

$\Leftrightarrow$ end $(f^S(t)) \in \phi_0^S(P)$,

$\Leftrightarrow E^S \models P(t)$.

H(t,h). $E^N \models^N H(t,h) \Leftrightarrow f^N(t) = f^N(h)$,

$\Leftrightarrow Kf^N(t) = Kf^N(h)$ by K3, both directions,

$\Leftrightarrow f^S(t) = f^S(h)$,

$\Leftrightarrow E^S \models H(t,h)$.

$\sim p$, $p \vee q$. Trivial.

$t \leq s$. $E^N \models^N t \leq s \Leftrightarrow f^N(t) \leq f^N(s)$,

$\Leftrightarrow Kf^N(t) \leq Kf^N(s)$ by K3,

$\Leftrightarrow f^S(t) \leq f^S(s)$,

$\Leftrightarrow E^S \models t \leq s$.

$t < h$. Similar to $t \leq s$.

$\underline{\exists tp}$. $E^N \models^N \exists tp$ iff $(\exists \tau \in pre(\pi^N))((E^N)^\tau_t \models^N p)$.

But the standard environment associated with $(E^N)^\tau_t$ is $(E^S)^{K\tau}_t$, so by induction

(**) $E^N \models^N \exists tp \Leftrightarrow (\exists \tau \in pre(\pi^N))((E^S)^{K\tau}_t \models p)$.

If τ is a finite legal prefix of some member of π^N, then by K1 and K3, $\tau' = K\tau$ is a finite legal prefix of some member of π^S. Hence

$$E^N \models^N \exists tp \Rightarrow (\exists \tau' \in pre(\pi^S))\ ((E^S)^{\tau'}_t \models p)$$
$$\Rightarrow E^S \models \exists tp.$$

Conversely, every $\tau' \in pre(\pi^S)$ is $K\tau$ for some τ in $pre(\pi^N)$, so,

$$E^S \models \exists tp \Rightarrow (\exists \tau' \in pre(\pi^S))\ ((E^S)^{\tau'}_t \models p),$$
$$\Rightarrow (\exists \tau \in pre(\pi^N))\ ((E^S)^{K\tau}_t \models p),$$
$$\Rightarrow E^N \models^N \exists tp \qquad \text{by (**)}.$$

$\underline{\exists hp}$. Similar to $\exists tp$, using π in place of $pre(\pi)$.

∎

3.4. A lower bound for GPL

We show that $L(\mathbb{N}, \leq, P)$, the theory of the natural numbers under the usual order $\leq$ with a monadic uninter-

preted predicate P, is embedded in N-GPL. $L(\mathbb{N},\leq,P)$ is nonelementary by Meyer [M74].

Syntactically, $L(\mathbb{N},\leq,P)$ is a subset of GPL, with integer variables corresponding to stage variables. Stages can be made to correspond to integers, with prefix corresponding to $\leq$ on $\mathbb{N}$, by quantifying stage variables relative to a particular infinite path h. The existence of such a path in a GPL model is ensured by $(\exists h)(\forall t\leq h)(\exists s\leq h)(t<s)$. In nonstandard semantics, any interpretation of P by an $L(\mathbb{N},\leq,P)$-model can be duplicated by a GPL model. Further details of the embedding are left to the reader.

Theorem 3.2. The validity (equivalently the satisfiability) problems for GPL is not elementary recursive. ∎

3.5. Closed GPL

Though we do not know whether GPL is decidable, we can show that GPL over a particular class of processes, the closed processes, is decidable. Moreover, there are some properties which can be expressed in GPL for closed processes, but which may not be expressible in GPL for arbitrary processes. Hence, for some applications, closed GPL may be more suitable than GPL.

Definition. A process π is **closed** if for every ascending prefix chain $\tau_1<\tau_2<\ldots$, each of whose members

is in pre(π), the limit of the sequence $\tau_1, \tau_2, \ldots$ is in π.

Example 1. If π is the set of paths in a finite branching, but possibly infinite depth, tree, then by König's lemma π is closed.

Example 2. $\pi = \{(0, <0\to0>^i <0\to1><1\to1>^\omega) : i \geq 0\}$ is not closed, for $(0, <0\to0>^i)$ is in pre(π) for all i, but $(0, <0\to0>^\omega)$ is not in π.

In Chapter 1 we described an interpreter which evaluates processes. Whenever the interpreter encounters a block on one path, it tries another path. Suppose we run the interpreter on the non-closed process $\pi = \{(0, <0\to0>^i) : i>0\}$. The interpreter would constantly choose longer and longer paths; in fact, it would behave as if it were following the fictitious path $(0, <0\to0>^\omega)$. Of course, that path is in the closure of π. Allowing the interpreter to change paths at will in effect closes the process being evaluated. Thus, in closed processes, our notion of an interpreter makes sense. In non-closed processes we must be very careful.

The usual sequential program constructs if-then-else, while-do, and sequencing preserve closed processes. However, as mentioned on page 9 , a fair concurrency operator does not preserve closed processes. Closed GPL, or C-GPL, can be thought of as the theory of sequential processes.

Sequential processes are often deterministic. If those processes are also assumed to be closed, then C-GPL can be made into

a logic of deterministic sequential processes, for "π is deterministic" can be expressed in GPL as $(\forall t_1,t_2,t_3 \in pre(\pi))(t_2=succ(t_1) \wedge t_3=succ(t_1) \supset t_2=t_3)$.

There are satisfiable GPL formulas which are not satisfied by any environment whose process is closed. An example is a formula which expresses

1) π contains no infinite paths, and

2) for every stage in $pre(\pi)$, there is a longer stage in $pre(\pi)$.

Clearly, no closed process can satisfy both (1) and (2). But the process $\{(0,\langle 0\rightarrow 0\rangle^i): i \geq 0\}$ does satisfy both of them. (1) and (2) are written in GPL as

1) $(\forall h)(\exists t \leq h)(\forall s \leq h)(s \leq t)$,

2) $(\forall t)(\exists s)(t<s)$.

We have just proved

Theorem 3.3. $SAT(C\text{-}GPL) \neq SAT(GPL)$. ∎

This is in contrast to SOAPL, where any satisfiable formula is satisfied by a closed model.

C-GPL may in a sense be more expressive than GPL. Suppose we wish to write "π must terminate," in the sense that π can't run forever, and π can't block, assuming an interpreter which tries all alternatives whenever a block is encountered. In C-GPL, "π must terminate" is expressed by the GPL equivalent of the following two sentences:

1. π contains no infinite paths.

2. If path $h \in \pi$ blocks at stage t, then there is a path $h' \in \pi$, with t as a prefix, which does not block at stage t.

In C-GPL, sentence (2) is $(\forall h)(\forall t \leq h)(H(t,h) \vee (\exists h' \geq t)(\exists t' \leq h')(t<t'))$. Sentences (1) and (2) can of course be written in GPL, but they no longer have the desired meaning. For the process $\pi = \{(0,<0\rightarrow 0>^{i}<1\rightarrow 1>):i \geq 0\}$ satisfies both (1) and (2), although π contains no terminating paths. The reader should be able to convince himself that (1) and (2) do express "π must terminate" when π is closed. There does not appear to be any way to express "π must terminate" which has the desired meaning for all processes.

There is an algorithm for deciding satisfiability of formulas in C-GPL. Following Parikh [Pa78], we embed *nonstandard* C-GPL into SnS, the second order theory of n successors (Rabin [R69]). SnS is recursive by Rabin, and nonelementary by Meyer [M74].

SnS describes strings over a finite alphabet $\Sigma=\{s_1,\ldots,s_n\}$. There are two kinds of variables: first order variables, ranging over Σ^*, and second order variables, ranging over $P(\Sigma^*)$. In addition to variables and the symbols $\exists$, $\sim$, $\vee$, there are primitive formulas for relating variables:

1. $x = y \cdot s_i$, where x and y are first order variables,

and • is concatenation.

2. $x \varepsilon X$, where x is first order, and X is second order.

Theorem 3.4. SAT(N-C-GPL) is recursive.

Corollary 3.5. SAT(C-GPL) is recursive.

Proof. In the proof of theorem 3.1. if π^N is closed, then so is π^S. ∎

Corollary 3.6. Deterministic C-GPL is recursive.

Proof. We showed above how to express "π is deterministic" in GPL. ∎

Proof of Theorem 3.4. The idea is to encode a non-standard structure into set variables in SnS. The structure $A = (U,\pi,\Phi_0,\phi_0)$ is coded as follows:

1. Let $U = \{u_1,u_2, \ldots\}$. u_i is coded as $\text{¢}a^i\$$.

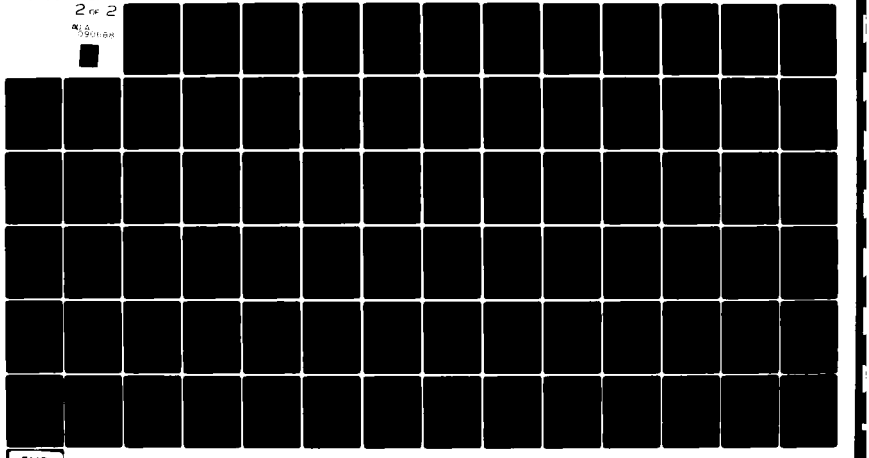
2. The finite paths of π can be coded into a single set variable Π_f. A finite legal path is represented by a string in $(\text{¢}a^*\$)^+\cdot\underline{t}$, where $\underline{t}$ is a special symbol flagging a terminated path. A blocked path is represented by the sequence of states up to the first block, followed by the special symbol $\underline{b}$.

3. Infinite paths are coded as limits of sets of finite paths. The set variable Π_i holds all finite legal prefixes of infinite paths in π. Because π is closed, the limits of infinite prefix chains of members of Π_i are exactly the infinite legal paths in π.

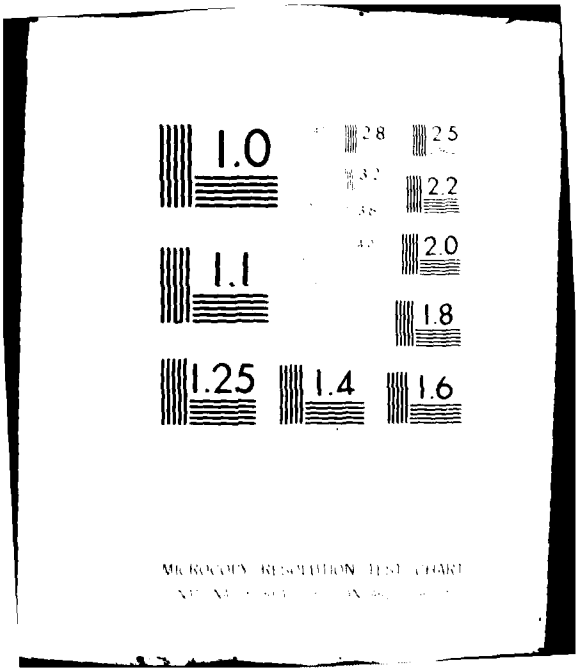
4. Let $\Phi_0 = \{P_1, \ldots, P_k\}$. For every $1 \leq i \leq k$ there is a set variable F_i which holds $\phi_0(P_i)$, the set of finite

AD-A090 688 WASHINGTON UNIV SEATTLE DEPT OF COMPUTER SCIENCE F/G 12/1
DECIDABILITY AND EXPRESSIVENESS OF LOGICS OF PROCESSES.(U)
AUG 80 K R ABRAHAMSON N00014-80-C-0221
UNCLASSIFIED TR-80-08-01 NL

2 OF 2
AD A090688



END
DATE FILMED
11-80
DTIC


1.0
2.8
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART

legal paths which satisfy P_i.

Before defining the translation Q:C-GPL→SnS, we list some useful abbreviations for SnS formulas.

1. The prefix relation $x \leq y$ on strings can be expressed in SnS as

$$x \leq y \equiv \forall X (x \in X \wedge \bigwedge_{i=1}^{n} (\forall z \in X)(\exists w \in X)(w = z \cdot s_i) \supset y \in X).$$

2. $x = y \equiv x \leq y \wedge y \leq x$.

3. $X \subseteq Y \equiv \forall x (x \in X \supset x \in Y)$.

4. singleton $(X) \equiv \forall x \forall y (x \in X \wedge y \in X \supset x = y) \wedge \exists x (x \in X)$.

5. ordered (X) = x is linearly ordered under prefix
$\equiv \forall x \forall y (x \in X \wedge y \in X \supset (x \leq y \vee y \leq x))$.

6. ascending $(X) = \forall x \exists y (x \in X \supset y \in X \wedge x < y)$.

7. infinitepath (X) = X represents a single infinite path $\equiv$ ordered $(X) \wedge$ ascending (X).

8. end $(x, s_i) = \exists y (y = x \cdot s_i)$.

9. If R is a regular expression over Σ, $x \in R$ can be expressed in SnS. See Parikh [Pa78].

10. in $\pi(X) =$ (singleton $(X) \wedge X \subseteq \Pi_f$) $\vee$
(infinitepath $(X) \wedge X \subseteq \Pi_i$).

11. inpre$\pi(x) = x \in (¢a*\$)^+ \wedge \exists y (x \leq y \wedge y \in \Pi_i \cup \Pi_f)$.

Let $t_1, t_2, \ldots$ be the stage variables, and $h_1, h_2 \ldots$ be the path variables. Associated with each t_i is a first order variable x_i. The value in x_i is always a member of $(¢a*\$)^+$. Associated with each h_i is a set variable X_i, which contains a single string, ending on $\underline{b}$ or $\underline{x}$,

when h_i is finite, and contains all finite prefixes of h_i when h_i is infinite. Define T:C-GPL → SnS inductively as follows:

$$T(P_j(t_i)) = x_i \in F_j.$$

$$T(H(t_i,h_j)) = \text{singleton}\ (X_j) \wedge \exists y(y \in X_j \wedge y = x_i \cdot \underline{t}).$$

$$T(t_i \underline{\leq} t_j) = x_i \underline{\leq} x_j.$$

$$T(t_i \underline{\leq} h_j) = \exists y(y \in X_j \wedge x_i \underline{\leq} y).$$

$$T(\sim p) = \sim T(p).$$

$$T(p \vee q) = T(p) \vee T(q).$$

$$T(\exists t_i p) = \exists x_i(\text{inpre}_\pi(x_i) \wedge T(p)).$$

$$T(\exists h_i p) = \exists X_i(\text{in}_\pi(X_i) \wedge T(p)).$$

Let $R = (\$a^*\$)^+$. Q: C-GPL → SnS is defined by

$$Q(p) = (\exists \Pi_f, \Pi_i, F_1, \ldots, F_k)$$
$$(\Pi_i \subseteq R \wedge \Pi_f \subseteq R \cdot (\underline{t} \cup \underline{b}) \wedge \bigwedge_i F_i \subseteq R$$
$$\wedge \text{ascending } (\Pi_i)$$
$$\bigwedge_i \text{inpre}_\pi(x_i)$$
$$\bigwedge_i \text{in}_\pi(h_i)\ )$$

<u>Claim</u>. $p \in$ SAT(N-C-GPL) iff Q(p) is true.

<u>Proof</u>. We have already explained how to obtain $\Pi_f, \Pi_i, F_1, \ldots, F_k$ from a structure. All of the conditions on $\Pi_f, \Pi_i, F_1, \ldots, F_k$ listed in Q(p) are easily seen to hold for the values obtained from a structure. It

is routine to show that those values also satisfy T(p), provided $(A,f) \models^N p$, and x_i and x_j are given the values associated with $f(t_i)$ and $f(h_j)$ respectively, for all i and j.

Conversely, the conditions on Π_f, Π_i, F_1, ..., F_k listed in Q(p) are sufficient to ensure that Π_f, Π_i, F_1, ..., F_k define a structure. The process of that structure is clearly closed. Again, it is routine to show that T(p) is true iff $(A,f) \models^N p$, where A is the structure defined by Π_f, Π_i, F_1, ..., F_k, and f assigns the values associated with x_i and x_j to t_i and h_j respectively, for all i and j. ∎

Theorem 3.7. SAT(C-GPL) is not elementary recursive.

Proof. The proof of theorem 3.2 requires only singleton processes, which are closed. ∎

3.6. GPL_M

In Chapter 4 we define a modal logic MPL, and show that MPL is decidable. GPL_M is a subset of GPL which is expressively equivalent to MPL over MPL environments, which are a subset of GPL_M environments. Decidability of GPL_M follows from the effectiveness of the embedding of GPL_M in MPL.

While $SAT(GPL_M)$ is not elementary recursive, SAT(MPL) is in $DTIME(2^{2^{cn}})$ for some constant c. Hence, even though

MPL and GPL_M have equal expressive power, MPL seems to be a more reasonable logic. The main purpose in studying GPL_M is to get a handle on just how powerful MPL is.

The GPL_M formulas are characterized by the following rules.

1. Every GPL_M formula has only one path variable h, though h may be repeatedly requantified.

2. Every subformula of the form $\exists h\ p$ of a GPL_M formula has exactly one free variable.

3. h can only be quantified relative to some stage variable, as $(\exists h \geq t)p$.

4. Every stage variable s can only be quantified beyond another stage t, and on path h, as $\exists s(t \geq s \geq h \wedge p)$.

GPL_M can be regarded as an extension of Gabbay et al.'s [GPSS80] future temporal logic (FTL). An FTL formula describes a particular path h. Path quantifiers are not allowed, and stage quantifiers range over h. In addition, every stage quantifier must have the form $(\exists s \geq t)$, where t is a distinguished stage variable.

GPL_M permits path quantifiers in certain settings. Wherever P(s) may appear in an FTL formula, $(\exists h \geq s)p$ may appear in a GPL_M formula, where only s is free in $(\exists h \geq s)p$. Thus a means is provided of considering all possible continuations from a given point on a path.

The restrictions made on GPL_M are superficially similar to those made by Parikh on SOAPL. However,

unlike SOAPL, whose every satisfiable formula, according to Parikh, is satisfied by a closed process, GPL_M contains formulas which are satisfied only by processes which are not closed. An example of such a formula is a modification of the one given for GPL on page 84. In abbreviated form it is:

1. $(\forall\, h \geq t)(\exists\, t', t \leq t' \leq h)(\forall\, t'', t \leq t'' \leq h)(t'' \leq t')$, and

2. $(\forall\, h \geq t)(\forall\, t', t \leq t' \leq h)(\exists\, h \geq t')(\exists\, t'', t' \leq t'' \leq h)(t' < t'')$.

In words,

1. Every member of π is finite, i.e., there is a maximal stage on every path.

2. For every stage of every path, there is a longer stage, possibly of a different path.

In Chapter 5 we show that, when programs are added to MPL, MPL can simulate SOAPL.

While closed processes are not sufficient for all GPL_M formulas, there is a different countable class of processes which is complete for GPL_M, in the sense that every satisfiable formula is satisfied by an environment whose process is a member of the class. That class is the class of LL-processes, defined in Chapter 4. There it is shown that LL-processes are complete for MPL.

GPL_M is a real restriction of GPL. Even without using path variables, we can write a GPL formula which is

not equivalent to any GPL_M formula in all environments. Such a formula is $D \equiv (\exists t_1, t_2)(t_1 \neq t_2)$. D is not a GPL_M formula, for t_1 and t_2 are not quantified relative to any path. D simply states that there are two distinct stages in pre(π). Consider the two structures A_1 and A_2, both with states {0,1}, $\phi_0 = \emptyset$ and $\phi_0 = \emptyset$, but with $\pi_1 = \{(0,\lambda)\}$ and $\pi_2 = \{(0,\lambda),(1,\lambda)\}$. Clearly A_1 does not satisfy D, while A_2 does. In GPL_M it is only possible to compare stages on the same path. But in A_1 and A_2 every path has only one prefix, namely itself, so $s \leq t$ is always true. Clearly, GPL_M cannot distinguish A_1 *from* A_2.

The proof of theorem 3.2 is easily modified to give the result that GPL_M is not elementary recursive.

Here is a summary of our results concerning GPL_M.

GPL_M is decidable but nonelementary.
GPL_M is strictly less expressive than GPL.
$SAT(GPL_M) \neq SAT(\text{closed } GPL_M)$.
GPL_M(MPL) with programs (see Chapter 5) is more expressive than SOAPL.

3.7. Open questions

As mentioned, we do not know whether GPL is decidable, although it is at best nonelementary. Although there are satisfiable GPL formulas which are not satisfiable by any closed process, there may be another countable class of

processes which is complete for GPL. A candidate is the class of LL-processes, which is at least complete for the subset GPL_M. Though we do not know whether LL-processes are complete for GPL, neither do we know of any satisfiable formula which is not satisfiable by an LL-process. Whether or not LL-processes are complete for GPL, they form an interesting class, and a study of GPL over them would be worthwhile.

We mentioned that we do not believe it is possible in GPL to state that π must terminate, though we have not proven it. Along the same lines, is it possible in GPL to state that π is closed? That π is an LL-process? (We conjecture "no" in both cases.)

Chapter 4

Modal Process Logic

In this chapter we define a process logic MPL, which is based on the use of certain operators to express properties of processes, rather than on explicit quantification of variables. We show that the expressive power of MPL exceeds that of some other proposed process logics, and is equal to the expressive power of GPL_M. Nevertheless, MPL has an elementary recursive decision problem. A major portion of this chapter is spent presenting an algorithm for deciding validity of MPL formulas, and proving that the algorithm works. The worst case running time of the algorithm is $O(2^{2^{cn}})$ on inputs of length n, for some constant c, and is far less on many inputs. Lastly, we derive a complete proof system for MPL from the decision algorithm.

4.1. An introduction to modal process logic.

The process logics studied in Chapter 3 all involved explicit variables and quantifiers. While quantifiers are powerful, they can be difficult to deal with, both on a formal and an intuitive level. An alternative is to make quantifiers implicit in certain operators. For example, rather than expressing global invariance as VtP(t), we could create an operator "GI," and simply write GI(P).

A modal logic can look very much like propositional calculus, with a few more operators, and can be handled in ways reminiscent of standard methods for dealing with propositional calculus. Proof systems for modal logic can be elegant, not having to deal with the problems arising from explicit variables.

Some languages which fit into the modal process logic class are described briefly below.

Hoare's logic [Ho69], based on the partial correctness assertion $p\{A\}q$, was one of the first to be studied. The partial correctness statement $p\{\pi\}q$ can be expressed in GPL as $(\forall h)(p(t) \supset (\forall t' \leq h)(H(t',h) \supset q(t))$, quite a long statement of a relatively simple property. Hoare Logic has the nice property that the statements it is designed to handle can be expressed concisely. An obvious shortcoming of Hoare Logic is that only partial correctness can be expressed.

Pratt's Dynamic Logic [Pr76] extends Hoare's Logic. Dynamic Logic is based on the operator [A]. [A]p holds at state u if p holds at every state where A could terminate, after being started in state u. The Hoare style partial correctness assertion $p\{A\}q$ can be expressed in Dynamic Logic as $p \supset [A]q$.

While Dynamic Logic is a termination oriented logic, more general properties of programs can be expressed in an augmented version of Dynamic Logic. We simply add

whatever new operators we desire. Pratt [Pr78] suggests, among others, a global invariance operator {A}p, meaning that p holds throughout the execution of program A.

Dynamic logic illustrates a general property of modal logics: a formula is not simply true or false, but is true at a given state, or, in the case of logics to follow, at a given stage on a given path.

Hoare style logic and Dynamic Logic are closely tied to programs as syntactic entities. But other languages have been studied which do not include programs, and so are more like GPL. A logic of Pnueli [Pn79] has two basic operators, G and X. Gp (generally p) holds at stage τ on path ψ if p(τ') holds for every $\tau \leq \tau' \leq \psi$. Xp holds at stage τ on path ψ if p holds for the successor of τ on ψ. Pnueli deals only with infinite paths, so there is no concern over whether the successor of τ exists. Gabbay, Pnueli et. al.[GPSS80] study a logic based on the operator <u>until</u> suggested by Kamp [K68], in terms of which both G and X can be expressed. They present a proof system for the logic of <u>until</u> and show that any statement which can be made using explicit time variables and quantifiers (or, in the GPL sense, stage variables and quantifiers) can be expressed in the logic of <u>until</u>. (p until q) holds at stage τ if q holds for some $\tau' \geq \tau$ on ψ, and p holds for every τ'' between τ and τ'.

Owicki [Ow78] suggests an operator <u>while</u>, p while q

meaning "p holds as long as q continues to hold." In view of the fact that p until q can be expressed in terms of while and X, it is of little concern which basis is chosen.

It is important to notice that the meaning of all of the operators G, X, until and while can be expressed in terms of stage quantifiers only. Hence any logic based solely on them must be severely restrictive in its use of path quantifiers. Lamport [L80], in his branching time logic, and Abrahamson [A79], go to the other extreme, forcing path quantifiers and stage quantifiers to appear in pairs.

Recently, Nishimura [N79] and Harel, Kozen and Parikh [HKP80] have extended the logic of until, introducing operators which stand for path quantifiers relative to certain programs. These logics were unknown to us when we developed MPL, and seem to extend the language of until in a slightly different direction. As programs are an integral part of those logics, we discuss them in Chapter 5.

4.2. The logic MPL

There are two types of operators in MPL, stage operators, which replace stage quantifiers, and path operators, which replace path quantifiers. Additionally, there is a special symbol H which replaces H(t,h). MPL can be

regarded as a syntactic restriction of GPL_M, and we give the GPL_M equivalent of each operator when defining it. The truth value of an MPL formula depends on a particular path and a particular stage on that path. Reflecting that are the two free variables h and t in the GPL_M equivalents of MPL formulas.

Stage operators

Stage operators are used to express properties of a given path. There are two primitive operators, Y and W, the rest being defined in terms of them.

1. Yp means "if there is a successor to t or h, then p holds there," and is equivalent to the GPL_M formula

$Yp \equiv (\forall s,\ t<s\leq h)((\forall r,\ t\leq r\leq h)(r\leq t \vee s\leq r) \supset p(s))$.

2. $Xp \equiv \sim Y \sim p$ means "there is a successor to t on h, and p holds there."

3. pWq (p <u>while</u> q) means "as long as q continues to hold beyond t on h, p continues to hold," and is equivalent to the GPL_M formula

$pWq \equiv (\forall s,\ t\leq s\leq h)((\forall r,\ t\leq r\leq h)(r\leq s \supset q(r)) \supset p(s))$.

4. $pBq \equiv \sim(\sim p\ W \sim q)$ (p <u>before</u> q) means "p holds at some stage $t' \geq t$, and q does not hold at any stage before or equal to t'."

5. $Gp \equiv pW$ true (<u>generally</u> p) means "p holds at every stage beyond t on h."

6. $Fp \equiv \sim G \sim p \equiv$ p B false (in the future p) means "p holds at some stage beyond t on h."

Although, as mentioned earlier, W and Y can both be expressed in terms of the single operator until, we find the two operators W and Y more convenient. Until is expressed in terms of W and Y as

p until $q \equiv X(Fq \wedge pW \sim q)$.

Path operators

We have already given compelling reasons for having path quantifiers in GPL. The same reasons are equally compelling for MPL. As a substitute for path quantifiers, we introduce the operator $\Box$, suggested by Michael J. Fischer [private conversation], and its dual $\Diamond$. $\Box$ universally quantifies a certain path variable h, and $\Diamond$ existentially quantifies h.

1. $\Box p$ is equivalent to the GPL_M formula $(\forall h \geq t)$ $p(t,h)$.

2. $\Diamond p \equiv \sim \Box \sim p$ is equivalent to the GPL_M formula $(\exists h \geq t)p(t,h)$.

4.3. Formal semantics of MPL

A formal semantics for MPL, independent of GPL_M, is as follows: An environment $E = (A,\psi,\tau)$ consists of a structure $A = (U,\pi,\Phi_0,\phi_0)$, a path $\psi \in \pi$, and a stage $\tau \leq \psi$. We write $\psi,\tau \models p$ for $(A,\psi,\tau) \models p$ when A is understood. Let $P \in \Phi_0$, and $p,q \in$ MPL.

1. $P \in MPL$; $\psi,\tau \models P$ iff end $(\tau) \in \phi_0(P)$.

2. $H \in MPL$; $\psi,\tau \models H$ iff $\psi=\tau$.

3. $\sim p \in MPL$; $\psi,\tau \models \sim p$ iff not $(\psi,\tau \models p)$.

4. $p \vee q \in MPL$; $\psi,\tau \models p \vee q$ iff $\psi,\tau \models p$ or $\psi,\tau \models q$.

5. $Yp \in MPL$; $\psi,\tau \models Yp$ iff $((\psi=\tau<u \rightarrow v>\psi'$ and $\tau<u \rightarrow v>$legal$) \Rightarrow \psi,\tau<u \rightarrow v> \models p)$.

6. $pWq \in MPL$; $\psi,\tau \models pWq$ iff for every legal τ', $\tau \leq \tau' \leq h$, $((\forall \tau'', \tau \leq \tau'' \leq \tau')$ $(\psi,\tau'' \models q)) \Rightarrow \psi,\tau' \models p$.

7. $\Box p \in MPL$; $\psi,\tau \models \Box p$ iff $(\forall \psi' \geq \tau, \psi' \in \pi)(\psi',\tau \models p)$.

We have already shown that MPL can simulate the operators G, X and until. Gabbay et. al. [GPSS80] describe a number of properties which can be expressed in terms of those operators, which we do not repeat here. MPL can of course express all of those properties. MPL can express properties not expressible with G, X, until and while alone. Lamport [L80] gives a language with two operators $\Box$ and $\rightsquigarrow$, and gives two different sematics for $\Box$ and $\rightsquigarrow$, which he calls the "linear time" semantics and the "branching time" semantics. Lamport shows that each version can express properties not expressible in the other version. MPL can simulate both versions. To avoid confusion, we rename Lamport's $\Box$ operator BOX.

Under linear time,

BOX $p \equiv Gp$,

$\rightsquigarrow p \equiv Fp$.

Under branching time,

BOXp ≡ □ Gp

~~p ≡ □Fp.

MPL can simulate PDL, provided programs are strongly restricted. Only A and A* are permitted, where A is a particular basic program. A PDL formula p is translated to MPL formula p′ by replacing

[A]q by □Yq,

and [A*]q by □Gq.

Given a PDL model for p which assigns to A the relation $\rho(A)$, we can find an MPL model for p′ whose process is $\pi = \rho(A)^{\omega}$, all infinite paths whose transitions are pairs in $\rho(A)$. Conversely, suppose A is an MPL model for p′ with process π. We define a PDL model with states pre(π), and $\rho(A) = \{(x,y): x,y \in pre(\pi), y=succ(x)\}$. To make basic formulas go through basically unchanged, we must use nonstandard MPL, rather than standard MPL. Nonstandard MPL is defined analogously to nonstandard GPL (see Chapter 3). As there is a simple embedding of MPL in GPL, the nonstandard-satisfiable formulas of MPL are just the standard satisfiable formulas.

As a consequence of the embedding of PDL over A and A* in MPL, Fischer and Ladner's [FL79] DTIME(c^n) lower bound on PDL applies to MPL as well.

The classical modal logics T, S4 and S5 are all embedded in MPL. Fischer and Ladner remark that T, S4

and S5 are embedded in PDL over A and A*. Let L be the modal operator "for all visible worlds." By the PDL simulation, it can be seen that in T, Lp is $p \wedge \Box Yp$. In S4, Lp is just $\Box Gp$. For our simulation of S5, we prefer to point out the similarity between the $\Box$ operator of MPL and L of S5. Let worlds correspond to members of π. The value of P(x) for $x \in \pi$ is determined by the value of P at the second state on x. Thus, to translate an S5 formula to MPL, replace L by $\Box$ and basic formula P by XP.

MPL can express absence of deadlock. An absence of deadlock statement must express that, whenever a path blocks, there is an alternative path which does not block in the immediate future. Termination is considered a normal condition.

π cannot deadlock = $\Box G(H \vee \Diamond X \text{ true})$.

4.4. Relation of MPL to GPL_M

Let GPL_{M1} be the GPL_M formulas with a single free stage variable.

GPL_{M1} formulas can be characterized by the following two rules.

1. If p is a TL_1 formula relative to path h (every quantifier has the form $(\exists s, t \leq s \leq h)$), then p is a GPL_{M1} formula.

2. If p(s) is a GPL_{M1} formula with only s (and possibly h) free, then GPL_{M1} is closed under substitution of $(\exists h \geq s)p(s)$ for P(s), where P is a basic predicate.

In this section we show that MPL and GPL_{M1} can express the same properties. Environments for GPL_M and MPL are almost the same, each consisting of a structure, a path, and a stage. Thus it makes sense to say $\psi,\tau \models p$, where p is a GPL_{M1} formula. The only difference is that in a GPL_M environment the stage need not be a prefix of the path, which it must in an MPL environment. We get around that by considering MPL environments only, saying that MPL formula p and GPL formula p' are equivalent if $E \models p$ iff $E \models p'$ for every MPL environment E.

Theorem 4.1. There is a recursive translation T from GPL_{M1} formulas to MPL formulas such that for every MPL environment E and every GPL_{M1} formula p, $E \models p$ iff $E \models T(p)$. Conversely, there is a recursive translation T' from MPL

formulas to GPL_{M1} formulas such that for every MPL environment E and MPL formula q, $E \vdash q$ iff $E \vdash T'(q)$.

Proof. Translation T' has already been given. To find T, we follow Nishimura [N80], who applies the results of Gabbay et. al. [GPSS80] to a logic similar to MPL. Let TL be the predicate calculus of a total order < with monadic uninterpreted predicates, and let TL_1 be the formulas of TL with at most one free variable.

Kamp [K68] shows that TL_1 is expressively equivalent to the logic L(u,s) of two operators, until and since, defined in terms of TL as

$$p \text{ until } q \equiv (\exists s>t)(q(s) \wedge \forall r(t<r<s \supset p(r)),$$
$$p \text{ since } q \equiv (\exists s<t)(q(s) \wedge \forall r(s<r<t \supset p(r)).$$

Although until can be expressed in terms of W and Y, since cannot, for since looks into the past from time t, while W and Y look only into the future. Gabbay et. al. show that the logic L(u) of until only is expressively complete for those formulas of TL which look only into the future. A future formula of TL_1 is a formula with one free variable t, and such that every quantifier in the formula has the form $(\forall s \geq t)$ or $(\exists s \geq t)$.

Theorem 4.2. (Gabbay et al.) There is a recursive translation F from the future formulas of TL_1 to L(u)

such that in every model, TL_1 formula p holds at time t iff F(t) holds at time t. ∎

The proof of theorem 4.2 can easily be modified to handle the termination formulas H(t) in TL_1 and H in L(u). Because W and Y can simulate until, we can replace L(u) by $\Box$-free MPL. Future-TL_1 is just GPL_{M1} without path quantifiers, the path h providing the time domain. However, in both L(u) and TL_1 basic predicates are interpreted over times, or stages, rather than over states. Thus L(u) is a subset of nonstandard MPL, and future-TL_1 is a subset of nonstandard GPL_{M1}. Theorem 4.2 can be modified as follows:

Theorem 4.3. There is a recursive translation F from path-quantifier-free (pqf) GPL_{M1} formulas to $\Box$-free MPL with the property that for every nonstandard structure $A = (U,\pi,\phi_o,\phi_o)$, every $\psi \in \pi$, every stage $\tau \leq \psi$, and every pqf GPL_{M1} formula p, $A,\psi,\tau \models^N p$ iff $A,\psi,\tau \models^N F(p)$. ∎

In order to prove theorem 4.1, we must extend F to all of GPL_{M1}. Translation T is simultaneously defined and proved to satisfy theorem 4.1 inductively on the length of p. To avoid confusion, $\models_G$ denotes truth in GPL_M, and $\models_M$ denotes truth in MPL. The superscript N is dropped from $\models$ for clarity, and $A,\psi,\tau \models p$ is abbreviated $\psi,\tau \models p$.

Suppose p has the form $(\exists h \geq t)a$. Define

$$T((\exists h \geq t)a) = \Diamond T(a).$$

Then

$$\psi,\tau \Vdash_G (\exists h \geq t)a$$
$$\Leftrightarrow (\exists \psi' \in \pi)(\psi' \geq \tau \wedge \psi', \tau \Vdash_G a),$$
$$\Leftrightarrow (\exists \psi' \in \pi)(\psi' \geq \tau \wedge \psi', \tau \Vdash_M T(a))$$

by induction,

$$\Leftrightarrow \psi,\tau \Vdash_M \Diamond T(a).$$

On the other hand, suppose that p = a does not begin with $(\exists h \geq t)$. Let F be the translation of theorem 4.3. Define $R:GPL_{Ml} \to pqf\ GPL_{Ml}$ by letting R(q) replace every maximal subformula $b = (\exists h \geq s)a'(s)$ of q by a new basic predicate $Q^b(s)$. Let $R':MPL \to MPL$ be the translation which replaces Q^b by T(b). T(a) is defined as

$$T(a) = R' \circ F \circ R(a).$$

Claim. For every ψ and τ, $\psi,\tau \Vdash_G a$ iff $\psi,\tau \Vdash_M T(a)$.

Proof. Let Δ be the set of Q-variables used by R on a and let $A' = (U,\pi,\phi_0 \cup \Delta,\phi_0')$ be the extension of A to Q-variables which assigns

$$\phi_0'(P) = \phi_0(P) \text{ for } P \in \phi_0,$$
$$\phi_0'(Q^b) = \{\tau: A,\psi_0,\tau \Vdash_G b\}$$

The truth of $b = (\exists h \geq s)a'(s)$ does not depend on ψ_0, so the choice of ψ_0 in the definition of $\phi_0'(Q^b)$ is arbitrary, and ϕ_0' is well defined.

$$A,\psi,\tau \models_G a$$
$$\Leftrightarrow A',\psi,\tau \models_G R(a) \qquad \text{from the definition of } \phi_0',$$
$$\Leftrightarrow A',\psi,\tau \models_M F\circ R(a) \qquad \text{by theorem 4.3.}$$

But for every ψ and τ,

$$A',\psi,\tau \models_M Q^b$$
$$\Leftrightarrow \tau \in \phi_0'(Q^b),$$
$$\Leftrightarrow A,\psi,\tau \models_G b$$
$$\Leftrightarrow A,\psi,\tau \models_M T(b) \qquad \text{by induction, for b is}$$

a subformula of a. Hence replacing Q^b by $T(b)$ cannot change the truth value of any formula. Thus

$$A,\psi,\tau \models_G a \Leftrightarrow A,\psi,\tau \models_M R'\circ F\circ R(a),$$
$$\Leftrightarrow A,\psi,\tau \models_M T(a),$$

which proves the claim.

All that is left to proving theorem 4.1 is to note that we have proved it for nonstandard structures, and standard structures are a special case of nonstandard structures. ∎

We note that, while MPL and GPL_M have the same expressive power over MPL environments, the validity problem for MPL is elementary recursive, while that for GPL_M is not. (The translation T given above is not elementary recursive. In fact, F is not.) That leads us to believe that MPL may be a more suitable

language if one is interested in verifying the validity of formulas. On the other hand, there may be interesting statements which can be made concisely in GPL_M, but can only be expressed by very long MPL formulas, though we know of no such statements.

4.5 Decidability of MPL

This section proceeds as follows: First, we define a structure called an LL-graph (LL stands for limited looping). For each LL-graph, we define an associated MPL structure. An LL-graph is a finite representation of its associated MPL structure, the structure possibly having both infinite paths and infinitely many paths. Not all structures can be represented by LL-graphs, for there are only countably many LL-graphs, and there are $\aleph_2$ processes. Nevertheless, the LL-graphs are enough for our needs.

Next, we describe the algorithm for deciding satisfiability of MPL formulas (or, equivalently, validity of MPL formulas, since p is valid iff $\sim p$ is not satisfiable.) Given a satisfiable formula p_o, the algorithm constructs an LL-graph $L(p_o)$, whose associated structure satisfies p_o. On the other hand, given a formula p_o which is not satisfiable, the algorithm noticeably fails to construct an LL-graph for p_o.

Finally, we prove that the algorithm has the properties claimed for it in the preceding paragraph, and that

it requires time $O(2^{2^{cn}})$ in the worst case.

4.5.1. LL-graphs

A common approach to establishing the decidability of a logic is to show that every satisfiable formula is satisfied by a model of bounded size. Then one way to decide if a given formula is satisfiable is simply to try every model up to a certain size. MPL structures can be infinite in three different ways: they can have infinitely many states, infinitely many paths, and paths of infinite length. While it is possible to make do with finitely many states, it is easy to write formulas which are satisfiable only by processes which either have infinitely many paths or at least one infinite path. For example:

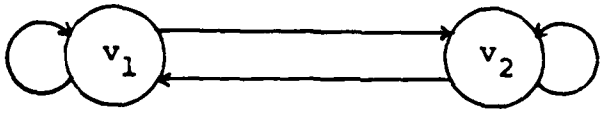
1) $\Diamond G X \text{true}$ forces π to contain at least one infinite path, and

2) $\Box G \Diamond X \text{true} \wedge \Box F Y \text{false}$ forces π to contain arbitrarily long paths, but no infinite paths, and so forces π to contain infinitely many paths.

An infinite process π can be represented as the set of paths in some finite directed graph. But there is a problem with that approach; the set of paths in a finite directed graph is closed, in the sense of C-GPL. But the satisfiable formula $\Box G \Diamond X \text{true} \wedge \Box F Y \text{false}$ mentioned above is not satisfiable by any closed process. Neverthe-

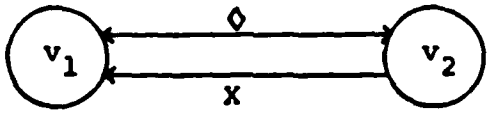
less, directed graphs can be used to represent processes, there being at least two ways to define a non-closed process from a directed graph.

1. Define the process associated with graph G to be the set of "fair" paths in G, where a path is fair provided, if it passes through node v infinitely often, then it passes through every node accessible from v infinitely often. For example, the set of fair paths in the graph



is not closed, for it does not contain the infinite path which remains in v_1 forever.

2. Let a directed graph have two different types of arcs, called ◊-arcs and X-arcs. Define the process associated with such a graph to be the paths which traverse finitely many ◊-arcs, but possibly infinitely many X-arcs. It is clear that the process associated with



is not closed.

We adopt both methods for LL-graphs. While the utility of ◊-arcs will become clear, the fairness condition is used mainly for technical reasons.

Definition. An LL-graph is a six-tuple $(V, A_{\Diamond}, A_X, V_B, \Phi_o, \phi_o)$ where

$(V, A_{\Diamond} \cup A_X)$ is a directed graph with vertices V, $\Diamond$-arcs $A_{\Diamond}$ and X-arcs A_X;

$V_B \subseteq V$ is a set of potential block vertices;

Φ_o is a finite set of basic formulas;

$\phi_o : \Phi_o \rightarrow P(V)$.

Additionally, an LL-graph must obey conditions LL1 and LL2. Let $\Diamond$ and X be the binary relations induced by $A_{\Diamond}$ and A_X respectively.

LL1. If $u \Diamond v$ then $v \Diamond u$. ($\Diamond$-arcs are bidirectional).

LL2. If $u \Diamond v$ then $u \in \phi_o(P)$ iff $v \in \phi_o(P)$ for every $P \in \Phi_o$.

The purpose of LL1 and LL2 will become clear later.

Definition. An arc-path in an LL-graph is a pair consisting of a start vertex and a sequence of zero or more (or infinitely many) arcs defining a connected path in L.

Definition. A route r in an LL-graph is an arc-path which satisfies R1-R3.

R1. r contains finitely many $\Diamond$-arcs.

R2. r does not end on a vertex with an X-arc leaving it.

R3. If r passes through vertex u infinitely often and there is a path of zero or more X-arcs from u

to v, then r passes through v infinitely often.

Definition. A simple route is a route which contains no $\Diamond$-arcs.

Note that there is at least one simple route starting at any vertex, which can be found by following X-arcs as long as they exist, using some fair system of choosing between X-arcs.

We are now in a position to define the MPL structure A_L associated with an LL-graph L. The states of A_L are the equivalence classes of the vertices of L under the equivalence relation $\Diamond^*$, the reflexive transitive closure of $\Diamond$. The paths of A_L are obtained from the routes in L. Given a route r, define the path $\bar{r}$ by

1) erasing all $\Diamond$-arcs in r,

2) replacing each X-arc $(u,v)_X$ by the transition $\langle \bar{u} \to \bar{v} \rangle$, where $\bar{u}$ is the equivalence class of u,

3) adding the transition $\langle \Lambda \to \Lambda \rangle$ to the end of $\bar{r}$ when $\bar{r}$ is finite and ends on a vertex in V_B.

The paths in A_L are the bars of the routes in L. Formally, given $L = (V, A, A_X, V_B, \phi_0, \phi_0)$, define $A_L = (\bar{V}, \pi, \phi_0, \bar{\phi}_0)$ by

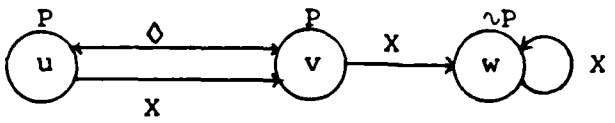
$\bar{u} = \{v \in V: u \Diamond^* v\}$,

$\bar{V} = \{\bar{u}: u \in V\}$,

$\pi = \{\bar{r}: r \text{ a route in } L\}$,

$\overline{\phi}_o(P) = \{\overline{u}: \ u \in \phi_o(P)\}$.

Example. The LL-graph



with $\phi_o(P) = \{u,r\}$ and $V_B = \emptyset$ represents a structure with two states, $\overline{u}$ and $\overline{w}$, and process $\pi = \{(\overline{u}, <\overline{u}\to\overline{u}>^i <\overline{u}\to\overline{w}> <\overline{w}\to\overline{w}>^\omega): \ i \geq 0\} \cup \{(\overline{w}, <\overline{w}\to\overline{w}>^\omega)\}$. $\overline{\phi}_o(P) = \{\overline{u}\}$.

4.5.2. The Decision Algorithm for MPL

Given formula p_o, the algorithm constructs a tableau for P_o, which is a generalization of an LL-graph. With each node u of a tableau there are associated two sets of formulas S_u and Z_u, which are used to guide the construction. The set V_B and function ϕ_o for a tableau are defined in terms of S_u by

$V_B = \{u: \ \sim H \in S_u\}$,

$\phi_o(P) = \{u: \ P \in S_u\}$.

Some of the nodes of a tableau are marked consistent, while others are marked inconsistent. The consistent subtableau T_c of T is obtained by deleting all inconsistent nodes and associated arcs from T. The tableau $T(P_o)$ constructed for P_o is designed to have the following property. Let A be the structure associated with $T_c(P_o)$, u be any node in

$T_c(p_o)$,q be any formula in $Z_u \supseteq S_u$, a_u be any finite arc-path in $T_c(p_o)$ ending on u, and r_u be any simple route starting at u. Then A, $\overline{a_u r_u}$, $\overline{a_u} \vdash q$. The bar of a_u is defined as for routes, with the exception that $\langle \emptyset \rightarrow \emptyset \rangle$ is not added to its end. By constructing $T(p_o)$ so that Z_v contains p_o for some node v, we see that, if v is consistent, then A, $\overline{a_v r_v}$, $\overline{a_v} \vdash p_o$, where $a_v = (v,\lambda)$ and r_v is a simple route starting at v, and hence p_o is satisfiable. Conversely, we show that if p_o is satisfiable, then Z_v contains p_o for some consistent node v.

The method of constructing $T(p_o)$ is similar to other tableau methods, such as that for classical modal logic [HC68], and PDL [Pr78]. We begin by setting T to the tableau consisting of a single node v_o, with $S_{v_o} = Z_{v_o} = \{p_o\}$. T does not yet obey the properties claimed for $T(p_o)$. In order to make T obey the claims, we perform transformations on T. Each transformation is intended to make one formula in one node hold for simple routes starting at that node, and accomplishes that goal either by adding new formulas or creating new nodes. For example, if S_u contains $\sim(p \vee q)$, a transformation replaces $\sim(p \vee q)$ by $\sim p$ and $\sim q$, in hope that future transformations will cause both $\sim p$ and $\sim q$ to be satisfied. If S_u contains

$p \vee q$, then a transformation causes u to split into two nodes u′ and u′′, one containing p, the other q. Transformations try to make both u′ and u′′ satisfy the claims, but need only succeed for one of them. Consistent nodes are ones on which transformations succeed. If S_u contains $\sim Yp$, a transformation creates a new node v, draws an X-arc from u to v, and places $\sim p$ (among other formulas) in S_v. If some alternative for v is consistent, then u is consistent. The hardest formulas to satisfy are the box formulas. The transformations first must reduce them to a standard form, which is a $\Box$ followed by a disjunction of one or more formulas, each starting either with Y or $\sim Y$. There are suitable transformations for formulas in standard form. $\sim\Box$ formulas are also reduced to standard form in order to avoid splitting transformations (such as that for $p \vee q$) from applying to nodes with $\Diamond$-arcs pointing to them, the reasoning being that if S_u contains $(P \vee \sim P)$, one alternative of u contains P, while the other contains $\sim P$. But condition LL2 requires that nodes linked by $\Diamond$-arcs satisfy exactly the same basic formulas. Splitting before drawing any $\Diamond$-arcs avoids that problem.

Transformations are applied until no more can be applied. At that point, consistency rules are invoked, causing some nodes to be marked inconsistent. When no more consistency rules apply, the construction is finished,

and $T = T(P_o)$.

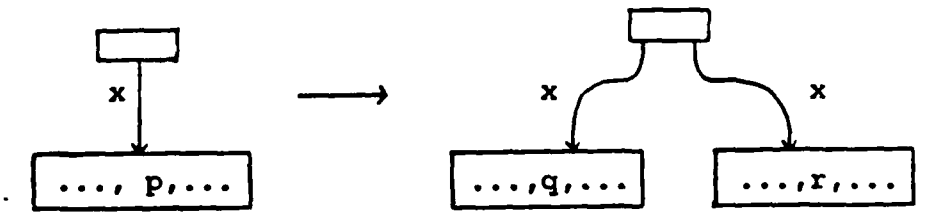
Transformations alter both S and Z sets. Set Z_u is a "history" set, containing every formula which was ever in S_u. In particular, $S_u \subseteq Z_u$. Some notation, similar to Pratt's [Pr78], will make transformations easier to write.

1. $p \rightarrow q, r$ means "if S_u contains p, then set $S_u := (S_u - \{p\}) \cup \{q,r\}$, and $Z_u := Z_u \cup \{q,r\}$."

2. $p \rightarrow q$ or r splits a node into two new nodes. If S_u contains p, replace u by two new vertices u′ and u′′, with

$$S_{u'} = (S_u - \{p\}) \cup \{q\}, \quad Z_{u'} = Z_u \cup \{q\},$$
$$S_{u''} = (S_u - \{p\}) \cup \{r\}, \quad Z_{u''} = Z_u \cup \{r\}.$$

If any X-arcs used to point to u, duplicate them for u′ and u′′ as shown below.



Due to the order in which transformations apply, no vertex with a $\Diamond$-arc pointing to it is ever split.

3. $\Box(a \vee p) \rightarrow \ldots$ In general, $\Box$ is followed by a disjunction of several terms, and only one of the terms is transformed. The disjunction $a \vee p$ is thought of as a set of formulas, one of whose members is p. Transformations

very similar to those for formulas outside the scope of $\Box$ apply to those inside the scope of $\Box$.

4. $p \Rrightarrow A$ is an abbreviation for two rules, one for p, the other for $\sim p$.

$p \Rrightarrow q,r$ represents $(p \rightarrow q,r)$ and $(\sim p \rightarrow \sim q \text{ or } \sim r)$.

$p \Rrightarrow q \underline{\text{or}} r$ represents $(p \rightarrow q \underline{\text{or}} r)$ and $(\sim p \rightarrow \sim q, \sim r)$.

Transformation Rules

The transformation rules are listed below. They are broken into five groups, transformations in group one having the highest priority, group two lower, etc. We assume that P_o is written using only basic formulas and the symbols $\sim$, $\vee$, H, $\Box$, W, Y, (,).

Group one

TR1. $\sim\sim p \rightarrow p$.

TR2. $p \vee q \Rrightarrow p \underline{\text{or}} q$.

TR3. $pWq \rightarrow \sim q \underline{\text{or}} (p, Y(pWq))$.

TR4. $\sim(pWq) \rightarrow (q, \sim p) \underline{\text{or}} (q, p, \sim Y(pWq))$.

Group two

TR5. $\Box(a \vee \sim\sim p) \Rrightarrow \Box(a \vee p)$.

TR6. $\Box(a \vee \sim(p \vee q)) \Rrightarrow \Box(a \vee \sim p), \Box(a \vee \sim q)$.

TR7. $\Box(a \vee pWq) \Rrightarrow \Box(a \vee \sim q \vee p), \Box(a \vee \sim q \vee Y(pWq))$.

TR8. $\Box(a \vee \sim(pWq)) \Rrightarrow \Box(a \vee q), \Box(a \vee \sim p \vee \sim Y(pWq))$.

Group three

TR9. $\Box(a \vee P) \Rrightarrow \Box a \underline{\text{or}} P$ for $P \in \Phi_o$.

TR10. $\Box(a \vee \sim P) \Rightarrow \Box a$ or $\sim P$ for $P \in \Phi_o$.

TR11. $\Box(a \vee \Box p) \Rightarrow \Box a$ or $\Box p$.

TR12. $\Box(a \vee \sim \Box p) \Rightarrow \Box a$ or $\sim \Box p$.

(If a is empty, $\Box a$ is false.)

Group five Rules for drawing X- and $\Diamond$-arcs.

TR14. a) If S_u contains either $\sim Yp$ or $\Box(\sim Yp_1 \vee \ldots \vee \sim YP_k)$, create a new vertex v, and draw an X-arc from u to v.

b) If part (a) results in a new vertex v, set

$$S_v = Z_v = \{p: Yp \in S_u\}$$
$$\cup \{\sim p: \sim Yp \in S_u\}$$
$$\cup \{(p_1 \vee \ldots \vee p_k \vee \sim q_1 \vee \ldots \vee \sim q_k),$$
$$\Box(p_1 \vee \ldots \vee p_k \vee \sim q_1 \vee \ldots \vee \sim q_m):$$
$$\Box(Yp_1 \vee \ldots \vee Yp_k \vee \sim Yq_1 \vee \ldots \vee \sim Yq_m)$$
$$\in S_u\}.$$

TR15. If S_u contains $\sim \Box(Yp_1 \vee \ldots \vee Yp_k \vee \sim Yq_1 \vee \ldots \vee \sim Yq_m)$ and there is no node v such that $u \Diamond^* v$ and S_v contains $\sim Yp_1, \ldots, \sim Yp_k, Yq_1, \ldots, Yq_m$ then create a new node v, draw a bidirectional $\Diamond$-arc between u and v, and set

$$S_v = Z_v = \{\sim Yp_1, \ldots, \sim Yp_k, Yq_1, \ldots, Yq_m\}$$
$$\cup \{P: P \in S_u \text{ and } P \in \Phi_o\}$$
$$\cup \{\sim P: \sim P \in S_u \text{ and } P \in \Phi_o\}$$
$$\cup \{\Box p: \Box p \in S_u\}.$$

TR16. Add $\sim\Box$ formulas to S and Z sets as required

to make the following true.

a) If $Yp_1, \ldots, Yp_k, \sim Yq_1, \ldots, \sim Yq_m$ are all of the Y and $\sim Y$ formulas in S_u, and u $\Diamond^*$ v for some v, then S_v contains $\sim \Box (\sim Yp_1 \vee \ldots \vee \sim Yp_k \vee Yq_1 \vee \ldots \vee Yq_m)$.

b) If u $\Diamond^*$v, then S_u and S_v contain the exact same $\sim\Box$ formulas.

(It is easy to show that TR16 cannot cause any other transformations to apply, or affect any consistency rules. Hence the algorithm works just as well without TR16. However, the correctness proof is simplified by having the redundant formulas which TR16 adds.)

A quick inspection of TR3 and TR14 shows that the transformations given so far can continue to create new vertices forever. However, after some time, the new vertices will be identical to previously constructed vertices. The filtration rule merges similar vertices. Filtration should be performed before group five rules, to prevent the creation of new nodes.

Group four (Filtration.)

TR13. If $S_u = S_v$ up to associativity and commutativity of $\vee$, delete u, and send any arcs which point to u to v instead. Set $Z_v := Z_v \cup Z_u$.

Consistency rules

C1. If Z_u contains both p and $\sim p$, then u is inconsistent.

C2. If S_u contains H and uXv for some v, then u is inconsistent.

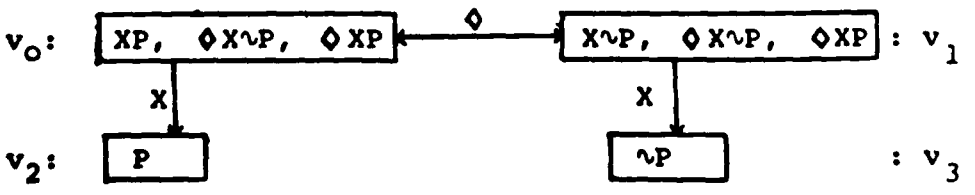
C3. If u $\Diamond$ v and v is inconsistent, then u is inconsistent.

C4. If there is some v such that uXv and every such v is inconsistent, then u is inconsistent.

C5. If Z_u contains $\sim(pWq)$ and for every consistent node v which is reachable from u by a path of zero or more X-arcs, Z_v contains p, then u is inconsistent.

The order in which the consistency rules apply makes no difference. It can be shown that a weaker version of C1, which only looks for a basic formula and its negation, is sufficient. The present version simplifies proofs.

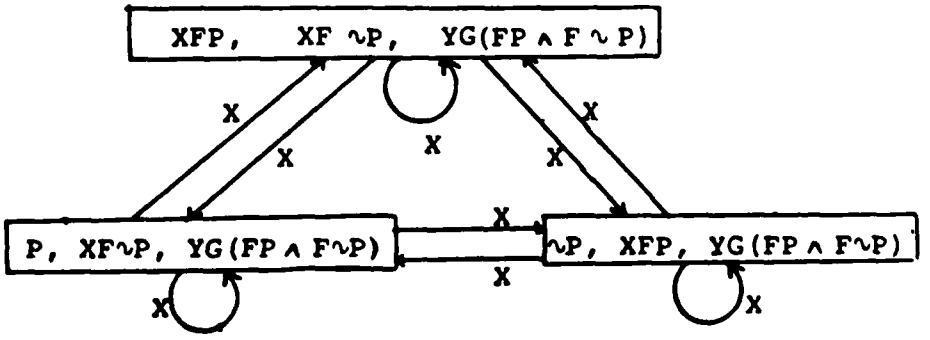
Example 1. The tableau constructed for $XP \wedge \Diamond X\sim P$ ($\equiv \sim (\sim Y\sim P \vee \Box\sim YP)$) is drawn below. We use X and $\Diamond$ freely to abbreviate $\sim Y\sim$ and $\sim \Box \sim$.



All of the nodes are consistent. The sets listed are the S sets, which in this simple example equal the Z sets at

every node. The formula $\Diamond XP$ was added to v_o and v_1 by TR16. Notice that, if r_{v_1} is the simple route $(v_1,(v_1,v_2)_X)$, so that $\bar{r}_{v_1} = (\bar{v}_1,\langle\bar{v}_1\rightarrow\bar{v}_2\rangle)$, then $\overline{a\cdot r}_{v_1}$, $\bar{a} \Vdash XP$ for any a ending on v_1. On the other hand, if r is the non-simple route $(v_1, (v_1, v_3)_\Diamond\ (v_3, v_4)_X)$, so that $\bar{r} = (\bar{v}_1, \langle\bar{v}_1\rightarrow\bar{v}_4\rangle)$, then $\overline{ar}$, $\bar{a}$ does not satisfy XP. This example illustrates a second function of $\Diamond$-arcs, in addition to limiting loop traversals. For a formula such as $\Diamond p_1 \wedge \ldots \wedge \Diamond p_n$ to hold at a given state, there must in general be several different paths through that state. $\Diamond$-arcs provide a means of splitting a state into n different nodes, in such a way that node i is responsible for creating a path which satisfies p_i.

Example 2. The consistent subtableau constructed for $G(FP \wedge F\sim P)$ is drawn below.



It is clear that every fair route satisfies $G(FP \wedge F\sim P)$, although some unfair routes, such as the one which remains in the lower left-hand corner forever, do not. This example

shows that the fairness condition is required for this particular algorithm to work, though not that fairness is required for there to be an LL-graph satisfying every formula, for the graph whose only route alternates between two nodes, one containing P and the other $\sim P$, also satisfies $G(FP \wedge F\sim P)$. We know of no formula which seems to require fairness.

Example 2 reveals that this algorithm sometimes constructs non-closed processes to satisfy formulas which are satisfiable by closed processes. Thus the algorithm cannot be used directly to decide satisfiability of MPL formulas over closed processes. We do not know of any better means of deciding closed MPL than by translating to C-GPL, and testing there.

4.5.3. Correctness of the decision algorithm

Let $T(p_o)$ be the tableau constructed for p_o, let $T_c(p_o)$ be the consistent subtableau of $T(p_o)$, and let A be the associated structure. We begin by bounding the time spent constructing $T(p_o)$.

Theorem 4.4. If p_o has length n, then there are at most $2^{2^{5n}}$ nodes in $T(p_o)$ for $n \geq 2$.

Proof. Let $S(p_o)$ be the set of formulas of the forms

a) p, b) $\sim p$,

c) Yp, d) $\sim Yp$,

where p is a subformula of p_o. A simple induction shows that $S(p_o)$ has at most $4n$ members. It is not difficult to show that every formula in Z_v for every v has one of the forms

1) q_1,

2) $\Box(q_1 \vee \ldots \vee q_k)$,

3) $\sim\Box(q_1 \vee \ldots \vee q_k)$,

where $q_1, \ldots, q_k$ are members of $S(p_o)$. Thus there are no more than $4n + 2 \cdot 2^{4n}$ different formulas written in nodes, up to associativity and commutativity of $\vee$.

By the filtration rule, no two distinct vertices can contain the exact same formulas, so there are at most

$$2^{4n+2^{4n+1}} < 2^{2^{5n}}$$ different vertices. ∎

Theorem 4.5. SAT(MPL) is in DTIME $(2^{2^{cn}})$ for some constant c.

Proof. We leave it to the reader that $T(p_o)$ can be constructed in time polynomial in the number of nodes in $T(p_o)$ in the worst case. For example, no more than 2^{5n} transformations can apply to any given node, and all of the formulas in a node have length at most c^n for some c. ∎

The best lower bound we know of on the complexity of MPL is the single exponential time bound which follows from MPL's ability to efficiently simulate PDL over the programs A and A*. Fischer and Ladner [FL79] prove that PDL over A and A* is not in DTIME(c^n) for some c>1.

Theorem 4.6. The satisfiability problem for MPL is not in DTIME(c^n) for some c>1. ∎

In section 4.6, we present a proof system A for MPL. Our goal is to prove the following theorem.

Theorem 4.7. Let v be a node in $T(p_o)$, and let p_v be the conjunction of all formulas in S_v. The following three statements are equivalent.

1. $\sim p_v$ is valid.
2. $\sim p_v$ is provable in system A.
3. v is inconsistent.

The remainder of this section is devoted to proving (1) $\Rightarrow$ (3). (3) $\Rightarrow$ (2) and (2) $\Rightarrow$ (1) are deferred to section 4.6, where system A is defined.

First we show that correctness of the decision method is a corollary of theorem 4.7. Assume without loss of generality that p_o has the form Xp. (If p_o has any other form, we can test Xp_o, which is valid iff p_o is valid.) Then v_o is never changed, and $p_{v_o} = p_o$.

Corollary 4.8. p_o is satisfiable iff v_o is consistent. ∎

Before proving (1) ⇒ (3), we prove four small lemmas.

Lemma 4.9. In the completed tableau $T(p_o)$, if $P \in \phi_o(\sim P, \Box p, \sim\Box p$ respectively), is in S_u and $\bar{u} = \bar{v}$ then $P(\sim P, \Box p, \sim\Box p$ respectively) is in S_v.

Proof. Lemma 4.9 for $\sim\Box p$ follows from the action of TR16. For P, $\sim P$ and $\Box p$ it follows from the fact that TR15 copies basic formulas, their negations, and box formulas across $\Diamond$-arcs, and no new box formulas can be created after TR15 applies. ∎

Lemma 4.10. $T_c(p_o)$ is an LL-graph.

Proof. We must verify that $T_c(p_o)$ satisfies LL1 and LL2. LL1 holds because TR15 draws bidirectional $\Diamond$-arcs. If $u \Diamond v$, then

$u \in \phi_o(P) \Leftrightarrow P \in S_u$ by definition of ϕ_o for T,

$\Leftrightarrow P \in S_v$ by lemma 4.9,

$\Leftrightarrow v \in \phi_o(P)$,

which verifies LL2. ∎

Lemma 4.11. Let v be a consistent node, and let p_v be the conjunction of all formulas in S_v. Then for every

formula q in Z_v, $p_v \supset q$ is valid.

Proof. Formally, the proof proceeds by induction on the number of transformations which have applied, showing that lemma 4.11 holds at every intermediate stage in the construction of $T(p_o)$, as well as in $T(p_o)$. Informally, we only need to notice that each transformation TR1-TR12 replaces a formula by an equivalent or stronger formula. For example, TR2 deletes $p \vee q$ from S_v, but adds either p or q, each of which implies $p \vee q$. TR14-TR16 do not remove any formulas from S_v. TR13 merges v with v', creating a new node v'', with $Z_{v''} = Z_{v'} \cup Z_v$ and $S_{v''} = S_{v'} = S_v$, which clearly preserves lemma 4.11. ∎

Lemma 4.12.

a) If Z_u contains Yp ($\sim$Yp) and uXw, then Z_w contains p ($\sim$p).

b) If Z_u contains $\Box(Yp_1 \vee \ldots \vee Yp_k \vee \sim Yq_1 \vee \ldots \vee \sim Yq_m)$ and uXw, then Z_w contains $\Box(p_1 \vee \ldots \vee p_k \vee \sim q_1 \vee \ldots \vee \sim q_m)$ and $(p_1 \vee \ldots \vee p_k \vee \sim q_1 \vee \ldots \vee \sim q_m)$.

Proof. TR14 places the desired formula in a node which is an ancestor of w in the construction. The history set Z_w retains the formula. ∎

Theorem 4.13. Let A be the structure associated

with $T_c(p_o)$. Let v be a consistent node in $T(p_o)$, a_v be an arc-path in $T_c(p_o)$ ending on v, and r_v be a simple route in $T_c(p_o)$ starting at v. Then A, $\overline{a_v \cdot r_v}$, $\bar{a}_v \models p$ for every p in Z_v.

Corollary 4.14. ((1) $\Rightarrow$ (3)). If v is consistent, then p_v is satisfiable.

Proof. There is a simple route r beginning at any given node in $T_c(p_o)$, which can be found by following X-arcs as long as they exist, using some fair method of choosing between X-arcs. By theorem 4.13, A, $\bar{r}$, $\bar{\lambda} \models \bigwedge_{p \in Z_v} p$, which implies that p_v is satisfiable.

Proof of theorem 4.13. The proof is by induction on the order $\prec$ over formulas which makes $p \prec q$ if either p has fewer W symbols than q, or p and q have the same number of W symbols, and p is shorter than q. If $p \prec q$, we say that p is smaller than q. Each possible form of p is considered below. We generally write $\bar{r}_v \models q$ for $\overline{a_v \cdot r_v}$, $\bar{a}_v \models q$ for brevity. We say that v contains p when $p \in Z_v$.

P. $P \in Z_v \Leftrightarrow P \in S_v$ because P is not reduced,
$\Leftrightarrow v \in \phi_o(P)$ by definition of $\phi_o(P)$,
$\Leftrightarrow \bar{v} \in \bar{\phi}_o(P)$ by LL2,
$\Leftrightarrow \bar{r}_v \models P$.

$\sim$P. $\sim P \in Z_v \Rightarrow \sim(P \in Z_v)$ by C1,

$\Rightarrow \sim(\overline{r}_v \models P)$ by the proof for P,

$\Rightarrow \overline{r}_v \models \sim P$

H. $H \in Z_v \Rightarrow \sim \exists u(vXu)$ by C2,

and $\sim H \notin S_v$ by C1, since H is not reduced,

$\Rightarrow \overline{r}_v = (\overline{v}, \lambda)$

$\Rightarrow \overline{r}_v \models H$

$\sim$H. Suppose $\sim H \in Z_v$. Then either there is a consistent u such that vXu, or $r_v = (v,\lambda)$. In the former case $\overline{r_v} \models \sim H$ by R2, which prohibits r_v from ending on v. When $r_v = (v,\lambda)$, $\overline{r_v} = (\overline{v}, \langle \Lambda \rightarrow \Lambda \rangle)$, and $\overline{r_v} \models \sim H$.

$\sim\sim p$, $p \vee q$, $\sim(p \vee q)$. Trivial, using TR1, TR2.

pWq. Suppose pWq is in Z_v. Let the nodes on route r_v be, in order, $v=v_1$, v_2, ... Because r_v is simple, there is an X-arc from v_i to v_{i+1} for all $i \geq 1$, up to the end of r_v, if r_v is finite. By TR3, any node v_i containing pWq also contains either $\sim q$ or both p and Y(pWq). If the latter is the case, then by lemma 4.12, v_{i+1} must also contain pWq, provided v_{i+1} exists. By repeatedly applying TR3 and lemma 4.12, we see that either v_1, ..., v_k all contain both p and Y(pWq) for some $k \geq 0$, and v_{k+1} contains $\sim q$, or every v_i contains both p and Y(pWq). Let r_i be the suffix of r_v which starts at v_i, and $a_v \cdot r_v = a_i \cdot r_i$. If every v_i contains p, then by induction $\overline{a_i r_i}$, $\overline{a_i} \models p$ for all i, which forces $\overline{a_v \cdot r_v}$, $\overline{a_v} \models pWq$. If, on the other hand, $v_1, \ldots, v_k$ contain p and v_{k+1} contains $\sim q$, then again by induction $\overline{a_v \cdot r_v}$, $\overline{a_v} \models pWq$.

$\sim(pWq)$. Suppose v contains $\sim(pWq)$. By repeated application of TR 4 and lemma 4.12, as was done for pWq, we see that either every node on r_v contains q and p and $\sim Y(pWq)$, or every node up to some point contains q and p and $\sim Y(pWq)$, and the next node contains q and $\sim p$. In the latter case, by induction and the meaning of pWq, $\overline{r_v} \models \sim(pWq)$. In the former case r_v must be infinite, for by R2 r_v cannot end on a node with an X-arc leaving it. Since every node on r_v contains $\sim Y(pWq)$, TR14 draws an X-arc coming out of every node on r_v, and by consistency

rule C4, every node on r_v (all of which must be consistent) retains at least one of its X-arcs in T_c. Route r_v must pass through some node w infinitely often, so by the fairness of r_v, r_v passes through node u infinitely often for every u which is reachable from w by a path of X-arcs. But every node on r_v contains both p and $\sim(pWq)$. Hence node w is inconsistent by C5, violating the fact that r_v is a route in T_c.

Yp. Suppose v is consistent and contains Yp. By lemma 4.12, any node u reachable from v by a single X-arc contains p. Suppose $r_v = (v,u)_x r_u$. By induction, $\overline{r_u} \models p$, and so $\overline{r_v} \models Yp$.

Yp. If v contains Yp, then TR14(a) draws an X-arc coming out of v. By C4, there must remain an X-arc coming out of v in T_c. The rest is very similar to Yp.

$\sim \Box p$. Suppose v is consistent and contains $\sim \Box p$. By TR15, there is a u in T such that $v \Diamond u$, and by consistency rule C3, u must be consistent. TR15 places formulas $q_1, \ldots, q_n$ in p, with $q_1 \wedge \ldots \wedge q_k \equiv \sim p$, and each q_i is smaller than $\sim p$. If r_u is a simple route starting at u, then by induction $\overline{r_u} \models q_i$ for all i, so $\overline{r_u} \models \sim p$. Hence there is a path $\overline{r_u}$ in A starting at $\bar{u} = \bar{v}$ which satisfies $\sim p$, which implies $\overline{a_v r}$, $\overline{a_v} \models \sim \Box p$ for any r, in particular for r_v.

There is a special problem with the box formulas. Several different transformations may apply to

$\Box(a_1 \vee \ldots \vee a_n)$, although, in any actual construction, only one is chosen. For that reason, it is not technically correct to say, for instance, that if $\Box(a \vee \sim\sim p)$ is in Z_v, then $\Box(a \vee p)$ is in Z_v, for a may have been reduced first. But some disjunct a_i of $\Box(a_1 \vee \ldots \vee a_n)$ is reduced in v, and we may consider $\Box(a_1 \vee \ldots \vee a_n)$ to be of the form $\Box(b \vee a_i)$. Thus, when proving theorem 4.13 for $\Box(a \vee p)$, we may assume that p is immediately reduced in v.

$\underline{\Box(a \vee \sim\sim p)}$, $\underline{\Box(a \vee \sim(p \vee q))}$. Trivial inductions.

$\underline{\Box(a \vee P)}$, $\underline{\Box(a \vee \sim P)}$, $\underline{\Box(a \vee \Box p)}$, $\underline{\Box(a \vee \sim \Box p)}$. Each of these is routine, by the group three transformations and the valid formulas

1) $\Box(a \vee P) \equiv \Box a \vee P$,

2) $\Box(a \vee \sim P) \equiv \Box a \vee \sim P$,

3) $\Box(a \vee \Box p) \equiv \Box a \vee \Box p$,

4) $\Box(a \vee \sim \Box p) \equiv \Box a \vee \sim \Box p$.

Formulas (1) - (4) can be recognized as valid by realizing that formulas P, $\sim P$, $\Box p$ and $\sim \Box p$ are independent of the variable h quantified by $\Box$.

We have considered $\Box(a \vee b)$ for every form of b except pWq, $\sim(pWq)$, Yp and $\sim Yp$. $\Box(a \vee pWq)$ and $\Box(a \vee \sim(pWq))$ are hardest to handle, and are done last. After TR1-TR12 have been exhaustively applied to v, the only remaining $\Box$ formulas in S_v have the form $\Box(Yp_1 \vee \ldots \vee Yp_k \vee \sim Yq_1 \vee \ldots \vee \sim Yq_m)$ for $k,m \geq 0$.

$\Box(Yp_1 \vee \ldots \vee Yp_k \vee \sim Yq_1 \vee \ldots \vee \sim Yq_m)$. Let $b \equiv Yp_1 \vee \ldots \vee Yp_k \vee \sim Yq_1 \vee \ldots \vee \sim Yq_m$, and $c \equiv p_1 \vee \ldots \vee p_k \sim q_1 \vee \ldots \vee \sim q_m$. Using the valid equivalences

1) $Y(p \vee q) \equiv Yp \vee Yq \equiv Yp \vee Xq$,

2) $X(p \vee q) \equiv Xp \vee Xq$,

we can show that

1) $b \equiv Yc$ if $k > 0$,

2) $b \equiv Xc$ if $k = 0$.

Case 1. Assume $k > 0$, and $\Box b \in Z_v$ (so $\Box b \in S_v$, since TR1-TR12 do not alter $\Box b$). We must show that $\overline{a_v r_v}, \overline{a_v} \vdash \Box b$, that is, for every route r (not necessarily simple) starting at v, $\overline{a_v r}, \overline{a_v} \vdash b$. Let r consist of a sequence d of zero or more $\Diamond$-arcs going from v to u, followed by an X-arc from u to w, followed by r', i.e., $r = d \cdot (u,w)_x \cdot r'$. Then

$\Box b \in S_u$ by lemma 4.9,

$\Longrightarrow \Box c \in Z_w$ by lemma 4.12,

$\Longrightarrow \overline{a_v \cdot d \cdot (u,w)_x r_w}, \overline{a_v \cdot d \cdot (u,w)_x} \vdash \Box c$, by induction,

$\Longrightarrow \psi, \overline{a_v \cdot d \cdot (u,w)_x} \vdash c$ for every $\psi \geq \overline{a_v \cdot d \cdot (u,w)_x}$,

$\Longrightarrow \overline{a_v r}, \overline{a_v \cdot d \cdot (u,w)_x} \vdash c$ since $\overline{r} \geq \overline{d \cdot (u,w)_x}$,

$\Longrightarrow \overline{a_v r}, \overline{a_v} \cdot \langle \overline{u} \rightarrow \overline{w} \rangle \vdash c$,

$\Longrightarrow \overline{a_v r}, \overline{a_v} \vdash Yc$,

$\Longrightarrow \overline{a_v r}, \overline{a_v} \vdash b$.

Case 2. Assume $k = 0$ and $\Box b \in S_v$. By lemma 4.9, $\Box b$ is in S_u for every $\bar{u} = \bar{v}$. When TR14 sees $\Box b \equiv \Box(\sim Yq_1 \vee \ldots \vee \sim Yq_m)$ in S_u, it draws an X-arc coming out of u. If v is consistent, then by C3 u is consistent, and by C4 there must be an X-arc leaving u in T_c. Thus no route can end on any u equivalent to v, so for every route r starting at v, $\bar{r} \models X$ true. It remains to show that $\bar{r} \models Yc$, since $b \equiv Xc \equiv X$ true $\wedge$ Yc. That was done in case 1.

$\Box(a \vee pWq)$. We need to know something about the formulas to which $\Box(a \vee pWq)$ is ultimately reduced.

Lemma 4.15. Suppose $\Box(a_1 \vee \ldots \vee a_n \vee Y(pWq))$ is in Z_v. Then there are formulas $q_1, \ldots, q_\ell$ in S_v such that $q_1 \wedge \ldots \wedge q_\ell \supset \Box(a_1 \vee \ldots \vee a_n \vee Y(pWq))$ is valid, and every q_i either has no more W symbols than some a_j, or is $(Yb_1 \vee \ldots \vee Yb_k \vee \sim Yc_1 \vee \ldots \vee \sim Yc_m \vee Y(pWq))$ for some $b_1, \ldots b_k, c_1, \ldots, c_m$, $k, m \geq 0$, where each b_i and c_i is no larger than some a_j.

Proof. Group two transformations apply to $\Box(a_1 \vee \ldots \vee a_n \vee Y(pWq))$ to produce several formulas, $\Box(d_1^i \vee \ldots \vee d_{n_i}^i \vee Y(pWq))$ for $i = 1, \ldots, t$, and it is easy to show that each d_j^i is either some a_k which was not reduced, or is smaller than some a_k, or is Ye or $\sim Ye$, where e is no larger than some a_k. Group three transformations pull each d_j^i which does not begin with Y or $\sim Y$ outside of

the box, and further transformations on d^i_j cannot produce a formula with more W symbols than d^i_j (although they can produce longer formulas). Thus the size constraints of lemma 4.15 are satisfied. By lemma 4.11 and the fact that each formula reduces independently of the others by TR1-TR12, if $\Box(a_1 \vee \ldots \vee a_n \vee Y(pWq))$ reduces to $q_1, \ldots, q_\ell$, then $q_1 \wedge \ldots \wedge q_\ell \supset \Box(a_1 \vee \ldots \vee a_n \vee Y(pWq))$ is valid.

Suppose $\Box(a \vee pWq) \in Z_v$,

$\Box(a \vee pWq) \in Z_v$

$\Rightarrow$ $\Box(a \vee q \vee \sim p) \in Z_v$ by TR7,

$\Rightarrow$ $\overline{r_v} \vdash \Box(a \vee \sim q \vee p)$ by induction,

for $\Box(a \vee \sim q \vee p)$ has fewer W's than $\Box(a \vee pWq)$.

$\Box(a \vee pWq) \in Z_v$

$\Rightarrow$ $\Box(a \vee \sim q \vee Y(pWq)) \in Z_v$ by TR7

$\Rightarrow$ $q_1, \ldots, q_\ell \in S_v$,

where $q_1, \ldots, q_\ell$ are the formulas of lemma 4.12. Those q_i which have no more W symbols than $a \vee \sim q$ are satisfied by $\overline{r_v}$ by induction. All that is left is to show that, if $q_i = \Box(Yb_1 \vee \ldots \vee Yb_k \vee \sim Yc_1 \vee \ldots \vee \sim Yc_m)$, then $\overline{r_v} \vdash q_i$. For then

$\overline{r_v} \vdash \Box(a \vee \sim q \vee p) \wedge q_1 \wedge \ldots \wedge q_\ell$

$\Rightarrow$ $\overline{r_v} \vdash \Box(a \vee \sim q \vee p) \wedge \Box(a \vee \sim q \vee Y(pWq))$ by the fact that $q_1 \wedge \ldots \wedge q_\ell \supset \Box(a \vee \sim q \vee Y(pWq))$ is valid,

$\Rightarrow$ $\overline{r_v} \vdash \Box(a \vee pWq)$ by semantic implication.

Equivalently, we must show that for every route r (not necessarily simple) starting at v, $\overline{a_v r}$, $\overline{a_v} \vdash Yb_1 \vee \ldots \vee Yb_k \vee \sim Yc_1 \vee \ldots \vee \sim Yc_m \vee Y(pWq)$.

Claim. Suppose, by induction, that theorem 4.13 holds for all formulas smaller than $\Box(a \vee pWq)$. Let $f = (Yd_1 \vee \ldots \vee Yd_s \vee \sim Ye_1 \vee \ldots \vee \sim Ye_t)$ and $g = (d_1 \vee \ldots \vee d_s \vee \sim e_1 \vee \ldots \vee \sim e_t)$, where each d_i and e_i is no larger than either a or q, and suppose that S_u contains $\Box(f \vee Y(pWq))$. Then for every route r starting at u, $\overline{a_u r}$, $\overline{a_u} \vdash f \vee Y(pWq)$.

Lemma 4.15 asserts that each $q_i = \Box(f_i \vee Y(pWq))$ satisfies the conditions of the claim. Hence, by the claim, $\overline{a_u r}$, $\overline{a_u} \vdash f_i \vee Y(pWq)$ for every r starting at u, which is what we want.

Proof of the claim. The proof is by subinduction on the order over routes which makes $r_1 < r_2$ iff either r_1 has fewer $\Diamond$-arcs than r_2, or r_1 and r_2 have the same positive number of $\Diamond$-arcs, and r_1 has fewer X-arcs before its first $\Diamond$-arc than r_2. Since routes can have only finitely many $\Diamond$-arcs, the induction covers all routes.

Case 1. r has no $\Diamond$-arcs, i.e., r is simple. If r has no arcs at all, then $\overline{r}$ trivially satisfies Y(pWq), and the claim holds. Suppose $r = (u,w)_x r'$. By lemma 4.12, Z_w contains $g \vee pWq$. TR2 selects one of $d_1, \ldots, d_s, \sim e_1, \ldots, \sim e_t, pWq$ to be in Z_w. The selected formula must be smaller than $\Box(a \vee pWq)$, so, by the main induction hypothesis, $\overline{r'}$ must satisfy it, since r' is simple. Hence

$\overline{a_u(u,w)_x r'}$, $\overline{a_u(u,w)_x} \vdash g \vee pWq$,

$\Rightarrow$ $\overline{a_u r}$, $\overline{a_u} \vdash Y(g \vee pWq)$,

$\Rightarrow$ $\overline{\alpha_u r}$, $\overline{\alpha_u} \vdash f$.

Case 2. $r = (u,w)_\Diamond \cdot r$ begins with a $\Diamond$-arc.

Let $a_w = a_u(u,w)_\Diamond$.

$\Box(f \vee Y(pWq)) \in S_u$

$\Rightarrow \Box(f \vee Y(pWq)) \in S_w$ by lemma 4.9,

$\Rightarrow$ $\overline{a_w \cdot r'}$, $\overline{a_w} \vdash f \vee Y(pWq)$ by the subinduction hypothesis

$\Rightarrow$ $\overline{a_u \cdot r}$, $\overline{a_u} \vdash f \vee Y(pWq)$ since bar erases $\Diamond$-arcs.

Case 3. $r = (u,w)_x r'$, and r' contains a $\Diamond$-arc.

$\Box(f \vee Y(pWq)) \in S_u$

$\Rightarrow \Box(g \vee pWq) \in Z_w$ by lemma 4.12,

$\Rightarrow \Box(g \vee \sim q \vee p) \in Z_w$ by TR7

$\Rightarrow$ $\overline{a_w \cdot r_w}$, $\overline{a_w} \vdash \Box(g \vee \sim q \vee p)$ by the main induction hypothesis,

$\Rightarrow$ $\overline{a_w \cdot r'}$, $\overline{a_w} \vdash g \vee \sim q \vee p$ by semantic implication.

Also,

$\Box(f \vee Y(pWq)) \in S_u$

$\Rightarrow \Box(g \vee pWq) \in Z_w$ by lemma 4.12,

$\Rightarrow \Box(g \vee \sim q \vee Y(pWq)) \in Z_w$ by TR7,

$\Rightarrow$ $\overline{a_w \cdot r'}$, $\overline{a_w} \vdash g \vee \sim q \vee Y(pWq)$ by lemma 4.15 and the subinduction hypothesis.

By the validity of $(g \vee \sim q \vee p) \wedge (g \vee \sim q \vee Y(pWq)) \supset g \vee pWq$, we have

$$\overline{a_w r}', \overline{a_w} \models g \vee pWq$$

$$\overline{a_u r}, \overline{a_u} \models Y(g \vee pWq)$$

$$\overline{a_u r}, \overline{a_u} \models f \vee Y(pWq)$$

by distributing Y over $\vee$.

$\Box(a \vee \sim(pWq))$. The proof for this case is very similar to that for $\Box(a \vee pWq)$. The main difference is that, instead of $Y(pWq)$, we have $\sim Y(pWq)$, and must take into account in lemma 4.15 the possibility that k might be zero. The tedious proof is omitted.

4.6. Proof and Completeness

As is the case with other decision algorithms based

on tableaux, a complete proof system for MPL can be derived from the tableau method for MPL. The axioms and inference rules of such a system are listed below as system A.

System A

Axioms

A1. All substitution instances of propositional calculus tautologies.

A2. $\Box p \supset p$.

A3. $\Box(p \supset q) \supset (\Box p \supset \Box q)$.

A4. $\Diamond p \supset \Box \Diamond p$.

A5. $\Diamond P \supset \Box P$ for $P \in \Phi_0$,

A6. $H \supset Y$ false.

A7. $\Box Yp \supset Y \Box p$.

A8. $Yp \equiv (X \text{ true} \supset Xp)$.

A9. $Y(p \supset q) \supset (Yp \supset Yq)$.

A10. $Gp \supset Yp$.

A11. $G(p \supset Yp) \supset (p \supset Gp)$.

A12. $G(p \supset q) \supset (Gp \supset Gq)$.

A13. $Gp \supset pWq$.

A14. $pWq \equiv q \supset (p \wedge Y(pWq))$.

Rules of Inference

PA1. $p, p \supset q \vdash q$ (Modus Ponens).

PA2. $p \vdash \Box p$.

PA3. $p \vdash Gp$.

Verification of soundness of system A is left to the reader. The only axiom which is not obviously valid is A7. Moving the Y in front of the $\Box$ effectively decreases the range over which $\Box$ quantifies to those paths which make the same next transition as the current path.

Before proving system A complete, we list a few useful theorems of system A. We say that p is provable by PC from $q_1, \ldots, q_n$ if p follows from $q_1, \ldots, q_n$ and instances of A1 by Modus Ponens.

The reader familiar with the classical modal logic S5 will recognize axioms A1-A4, together with proof rules PA1 and PA2, as a complete proof system for S5. It follows that every substitution instance of an S5 theorem, where $\Box$ and $\Diamond$ are taken to be the S5 modalities, is an MPL theorem. Due to axiom A5, the converse is not true; that is, there are MPL theorems involving only $\Box$, $\Diamond$, $\sim$ and propositional variables which are not S5 theorems. MPL is prevented from collapsing into propositional calculus only by the operators Y and W. Theorems TA1, TA2 and TA3 are all proved in [HC68] for S5.

Theorem TA1.

a) $\vdash \Box\Box p \equiv \Box p$, b) $\vdash \Box\Diamond p \equiv \Diamond p$,

c) $\vdash \Diamond\Box p \equiv \Box p$, d) $\vdash \Diamond\Diamond p \equiv \Diamond p$.

Theorem TA2. $\vdash \Box(p \wedge q) \equiv \Box p \wedge \Box q$.

Theorem TA3. For $P \in \Phi_0$, p any formula,

a) $\vdash \Box(a \vee P) \equiv \Box a \vee P$,

b) $\vdash \Box(a \vee \sim P) \equiv \Box a \vee P$,

c) $\vdash \Box(a \vee \Box p) \equiv \Box a \vee \Box p$,

d) $\vdash \Box(a \vee \Diamond p) \equiv \Box a \vee \Diamond p$.

Theorem TA4.

a) $\vdash Y(p \vee q) \equiv Yp \vee Yq$,

b) $\vdash Y(p \vee q) \equiv Yp \vee Xq$,

c) $\vdash X(p \vee q) \equiv Xp \vee Xq$,

d) $\vdash Y(p \supset q) \supset (Xp \supset Xq)$,

e) $\vdash Y(p \wedge q) \equiv Yp \wedge Yq$,

f) $\vdash X(p \wedge q) \equiv Xp \wedge Yq$,

g) $\vdash X(p \wedge q) \equiv Xp \wedge Xq$,

h) $\vdash Xp \equiv Yp \wedge X$ true,

i) $\vdash X\sim p \equiv \sim Yp$.

Proof. Let PA4 be the derived inference rule $p \vdash Yp$ (from PA3, A10 and PA1).

1) $(Yp \wedge X \text{ true}) \supset Xp$ — A8, PC;

2) $Xp \supset \sim y\sim p$ — definition of Xp;

3) $Xp \supset \sim(X \text{ true} \supset X\sim p)$ — (2), A8, PC;

4) $Xp \supset X\text{true} \wedge \sim X\sim p$ — (3), PC;

5) $\sim X\sim p \equiv Y\sim\sim p$ — definition of X, PC;

6) $Y(\sim\sim p \equiv p)$ — (5), A1, PA4;

7) $Y\sim\sim p \equiv Yp$ — (6), A9 twice, PC;

8) $Xp \equiv Xtrue \wedge Yp$ (1), (4), (7), PC;

9) $Y((p \vee q) \supset (\sim p \supset q))$ (8), A1, PA4;

10) $Y(p \vee q) \supset (Y\sim p \supset Yq)$ (9), A9 twice, PC;

11) $Y(p \vee q) \supset (Xp \vee Yq)$ definition of Xp, (10), PC;

12) $Y(p \supset (p \vee q))$ (11), A1, PA4;

13) $Yp \supset Y(p \vee q)$ (12), A9, PC;

14) $Yq \supset Y(p \vee q)$ symmetry, (13)

15) $Yp \vee Yq \supset Y(p \vee q)$ (13), (14), PC;

16) $Xp \supset Yp$ (8), PC;

17) $Xp \vee Yq \supset Yp \vee Yq$ (15), (16), PC;

18) $Xp \vee Yq \equiv Yp \vee Yq \equiv Y(p \vee q)$ (11), (15), (17), PC;

19) $Y(p \wedge q \supset p)$ A1, PA4;

20) $Y(p \wedge q) \supset Yp$ (19), A9, PC;

21) $Y(p \wedge q) \supset Yq$ symmetry, (20);

22) $Y(p \supset (q \supset p \wedge q))$ A1, PA4;

23) $Yp \supset (Yq \supset Y(p \wedge q))$ (22), A9 twice, PC;

24) $Yp \wedge Yq \supset Y(p \wedge q)$ (22), PC;

25) $Yp \wedge Yq \equiv Y(p \wedge q)$ (20), (21), (24), PC.

The reader should have no difficulty proving those parts of theorem TA4 which are not lines above.

Theorem TA5. $\Box X p \supset X \Box p$.

Proof.

1)	$\Box (Xp \equiv X\text{true} \wedge Yp)$	TA4(h), PA2;
2)	$\Box Xp \equiv \Box(X\text{true} \wedge Yp)$	(1), A3 twice, PC;
3)	$\Box Xp \equiv \Box X\text{true} \wedge \Box Yp$	(2), TA2, PC;
4)	$\Box X\text{true} \supset X\text{true}$	A2;
5)	$\Box Xp \supset X\text{true} \wedge \Box Yp$	(3), (4), PC;
6)	$\Box Xp \supset X\text{true} \wedge Y \Box p$	(5), A7, PC;
7)	$X \Box p \equiv X\text{true} \wedge Y \Box p$	TA4(h);
8)	$\Box Xp \supset X \Box p$	(6), (7), PC.

Some definitions of sets and formulas in the tableau $T(p_o)$ make the completeness proof more concise.

$S_v' = \{q \in S_v:$ q has the form P, $\sim P$, $\Box p$ or $\sim \Box p$.

$S_v'' = S_v - S_v'$.

$S_v^X = \{p: Yp \in S_v\} \cup \{\sim p: \sim Yp \in S_v\}$

$\cup \{(a_1 \vee \ldots \vee a_k \vee \sim b_1 \vee \ldots \vee \sim b_m),$

$\Box(a_1 \vee \ldots \vee a_k \vee \sim b_1 \vee \ldots \vee \sim b_m)$:

$\Box(Ya_1 \vee \ldots \vee Ya_k \vee \sim Yb_1 \vee \ldots \vee \sim Yb_m) \in S_v\}$.

$p_v = \bigwedge_{q \in S_v} q.$

$p_v'' = \bigwedge_{q \in S_v''} q.$ $\qquad$ $p_v' = \bigwedge_{q \in S_v'} q.$

$z_v = \bigwedge_{q \in Z_v} q.$ $\qquad$ $p_v^X = \bigwedge_{q \in S_v^X} q.$

We now state four lemmas, then prove that system A is complete.

Lemma 4.16. For TR1-TR12,

a) If $p \rightarrow q,r$ is a transformation, then $\vdash (p \equiv q \wedge r)$;

b) If $p \rightarrow q$ or r is a transformation, then

$\vdash (p \equiv q \vee r)$.

Proof. Routine. ∎

Lemma 4.17. For every node v in $T(p_o)$ (consistent or inconsistent), $\vdash (p_v \supset z_v)$.

Proof. Lemma 4.17 is proven almost identically to lemma 4.11. Where that proof uses the fact that if p is transformed to q, then $q \supset p$ is valid, here we must use the fact that $q \supset p$ is provable. That follows from lemma 4.16. ∎

Lemma 4.18. Suppose there is an X-arc leaving node v, and $u_1, \ldots, u_n$ are all of the nodes in $T(p_o)$ (both consistent and inconsistent) which are reachable from v by an X-arc. Then $\vdash (p_v^X \supset (p_{u_1} \vee \ldots \vee p_{u_n}))$.

Proof. Nodes $u_1, \ldots, u_n$ were created by first creating a node u_o by TR14, and then splitting u_o by or type transformations. We show that at every stage in the reduction of u_o to $u_1, \ldots, u_n$, if $u_1', \ldots, u_k'$ are the present nodes, then $\vdash (p_v^X \supset (p_{u_1} \vee \ldots \vee p_{u_k}))$.

The base case, u_o, is trivial since TR14 sets $S_{u_o} = S_v^X$. As each transformation TR1-TR12 is applied, either a conjunct of $P_{u_i'}$ is replaced by provably equivalent conjuncts for some i, by lemma 4.16(a), or u_i' is split by an or type transformation into u_i'' and u_i'''. By lemma 4.16(b), $\vdash (P_{u_i''} \vee P_{u_i'''} \equiv P_{u_i'})$. Hence $\vdash (P_v^X \supset P_{u_1'} \vee \ldots \vee P_{u_{i-1}'} \vee P_{u_i''} \vee P_{u_i'''} \vee P_{u_{i+1}'} \vee \ldots \vee P_{u_k'})$ by PC. The formula added to v by TR16(a) is implied by formulas already in S_v. TR16(b) adds no formulas to the original node of a group of nodes connected by $\Diamond$-arcs, and only that original node can have an X-arc pointing to it. It is easy to see that the filtration rule preserves lemma 4.18. ∎

Lemma 4.19. Suppose there is a u such that vXu. Then $\vdash (p_v \supset X\ p_v^X)$.

Proof. Let the Y, $\sim$Y and $\Box$ formulas in S_v be $Ya_1, \ldots, Ya_k, \sim Yb_1, \ldots, \sim Yb_\ell, \Box(Yc_1^i \vee \ldots \vee Yc_{m_i}^i \vee \sim Yd_1^i \vee \ldots \vee \sim Yd_{n_i}^i)$, for i=1, ..., t. By definition $p_v^X =$

$\bigwedge_i a_i \wedge \bigwedge_i \sim b_i \wedge \bigwedge_i \Box(\bigvee_j c_j^i \vee \bigvee_j \sim d_j^i)$. Because every member of S_v is a conjunct in p_v, we have

$$\vdash (p_v \supset \bigwedge_i Ya_i \wedge \bigwedge_i \sim Yb_i \wedge \bigwedge_i \Box(\bigvee_j Yc_j^i \vee \bigvee_j \sim Yd_j^i)).$$

Theorem TA4 can be used to bring conjunctions and disjunctions of Y and $\sim$Y formulas under a single Y. Axiom A7 is used to move a Y outside of a $\Box$. We get

$$\vdash (p_v \supset Y(\bigwedge_i a_i \wedge \bigwedge_i \sim b_i \wedge \bigwedge_i \Box(\bigvee_j c_j^i \vee \bigvee_j d_j^i))),$$

$$\Rightarrow \vdash p_v \supset Yp_v^X.$$

Moreover, TR14 only draws an X-arc from v to u if either $\ell > 0$ or $m_i = 0$ for some i. In either case, TA4 and A2 permit us to prove the stronger form

$$\vdash p_v \supset Xp_v^X . \quad \blacksquare$$

We now finish the proof of theorem 4.7, proving (3) $\Rightarrow$ (2) and (2) $\Rightarrow$ (1).

Lemma 4.20. ((2) $\Rightarrow$ (1)) If $\sim p_v$ is provable then $\sim p_v$ is valid.

Proof. By soundness of system A. $\blacksquare$

Lemma 4.21. ((3) $\Rightarrow$ (2)) If v is inconsistent in $T(p_o)$, then $\vdash \sim p_v$.

Proof. The proof is by induction on the order in which nodes are marked inconsistent.

Case 1. Suppose v is marked inconsistent by C1. Then Z_v contains both p and $\sim p$. By lemma 4.17, $\vdash (p_v \supset \sim p \wedge p)$, so by PC, $\vdash \sim p_v$.

Case 2. Suppose v is marked inconsistent by C2. Then S_v contains H and there is an X-arc leaving v. By lemma 4.19, $\vdash p_v \supset Xp_v^X$, so by TA4(h) $\vdash p_v \supset X$ true. Using axiom A6 and PC, we have $\vdash \sim p_v$.

Case 3. Suppose v is marked inconsistent by C3. Then there must be a node u connected to v by a $\Diamond$-arc, and which is marked inconsistent earlier than v. Let $p_v' = a_1 \wedge \ldots \wedge a_n$ and $p_v'' = b_1 \wedge \ldots \wedge b_n$. Define $\bar{p}_v' = \sim a_1 \vee \ldots \vee \sim a_n$, and $\bar{p}_v'' = \sim b_1 \vee \ldots \vee \sim b_n$.

$\vdash \sim p_u$ by induction,

$\Rightarrow \vdash \sim(p_u' \wedge p_u'')$ by $p_u \equiv p_u' \wedge p_u''$,

(*) $\Rightarrow \vdash \bar{p}_u' \vee \bar{p}_u''$ by PC,

$\Rightarrow \vdash \Box(\bar{p}_u' \vee \bar{p}_u'')$ by PA2.

Every formula in $\bar{p}_u'$ either begins with $\sim\Box$ or $\sim\sim\Box$, or is $\sim P$ or $\sim\sim P$ for some basic formula P, by the definition of p_u'. $\sim\sim$ can be eliminated at step (*) by PC. By repeated application of theorem TA3,

$\vdash \bar{p}_u' \vee \Box \bar{p}_u''$.

By TR16(a), $\sim\Box \bar{p}_u''$ is in S_u', and so $\sim\sim\Box \bar{p}_u''$ is a disjunct in $\bar{p}_u$. PC eliminates duplicate disjuncts, giving

$\vdash \bar{p}_u'$

$\Rightarrow \vdash \sim p_u'$ by PC,

$\Rightarrow \vdash \sim p_v'$ by lemma 4.9,

$\Rightarrow \vdash \sim p_v$ by $p_v \equiv p_v' \wedge p_v''$.

Case 4. Suppose v is marked inconsistent by C4. Then there is an X-arc leaving v, and all of the nodes $u_1, \ldots, u_n$

which are pointed to by X-arcs from v are marked inconsistent before v.

$\bigwedge_i$ (u_i inconsistent)

$\Rightarrow$ $\bigwedge_i \vdash \sim p_{u_i}$ by induction,

$\Rightarrow$ $\vdash \bigwedge_i \sim p_{u_i}$ by PC,

$\Rightarrow$ $\vdash \sim p_v^X$ by lemma 4.18,

$\Rightarrow$ $\vdash G \sim p_v^X$ by PA3,

$\Rightarrow$ $\vdash Y \sim p_v^X$ by A10,

$\Rightarrow$ $\vdash \sim X\ p_v^X$,

$\Rightarrow$ $\vdash \sim p_v$ by lemma 4.19.

Case 5. Suppose v is made inconsistent by C5. Let $v_1, \ldots, v_k$ be all of the nodes (including v itself) which are reachable from v by a path of zero or more X-arcs, and which are consistent when C5 applies to v. We may assume that C1 is applied wherever possible before C5 is used. We can show that every v_i has an X-arc leaving it. For, in order for C5 to apply, every Z_{v_i} must contain both $\sim(pWq)$ and p. If $\sim(pWq)$ is transformed to q,p and $\sim Y(pWq)$ by TR4, then v_i must have an X-arc leaving it. On the other hand, if $\sim(pWq)$ is transformed to q and $\sim p$, then v_i is inconsistent by C1. Let $v_1^i, \ldots, v_{m_i}^i$, $m > 0$, be all of the nodes for which $v_i\ X\ v_j^i$, for $i = 1, \ldots, k$, $j = 1, \ldots, m_i$. We write P_i for P_{v_1} and P_j^i for $P_{v_j^i}$

(1) $\vdash (P_i \supset X\ P_i^X)$ by lemma 4.19;

(2) $\vdash (p_i^X \supset \bigvee_j p_j^i)$ by lemma 4.18;

(3) $\vdash Y(p_i^X \supset \bigvee_j p_j^i)$ by (2), PA3, A10;

(4) $\vdash X\, p_i^X \supset X \bigvee_j p_j^i$ by (3), TA4(d);

(5) $\vdash p_i \supset X \bigvee_j p_j^i$ (1), (4), PC;

(6) $\vdash \bigvee_j p_j^i \supset \bigvee_i p_i$ by PC;

(7) $\vdash X \bigvee_j p_j^i \supset X \bigvee_i p_i$ by (6), PA3, A10, TA4;

(8) $\vdash p_i \supset X \bigvee_i p_i$ by (5), (7), PC.

But (8) holds for every i, so all can be combined by PC to give

(9) $\vdash \bigvee_i p_i \supset X \bigvee_i p_i$.

Let $q \leftarrow \bigvee_i p_i$.

(10) $\vdash q \supset Yq$ by (9), TA4(h);

(11) $\vdash G(q \supset Yq)$ by (10), PA3;

(12) $\vdash q \supset Gq$ by (11), A11.

Z_{v_i} contains p for all i, so by lemma 4.17

(14) $\vdash (p_i \supset p)$ for all i;

(15) $\vdash (q \supset p)$ by (14), PC;

(16) $\vdash G(q \supset p)$ by (15), PA3,

(17) $\vdash Gq \supset Gp$ by (16), A12;

(18) $\vdash p_i \supset G_p$ by p_i q, (12), (17), PC;

(19) $\vdash p_i \supset pWq$ by A13.

Choosing $v_i = v$, we see that

(20) $\vdash p_v \supset pWq$.

But $\sim(pWq)$ is in Z_v, or C5 wouldn't apply. By lemma 4.16

(21) $\vdash p_v \supset \sim(pWq)$,

and, combining (20) and (21) we have

$\vdash \sim p_v$.

Theorem 4.22 (completeness) If p is valid then p is provable in system A.

Proof. Let $u_1, \ldots, u_n$ be all of the nodes in $T(X\sim p)$ which are reachable from v_o by an X-arc. v_o is not changed when it contains $X\sim p$, so $p_{v_o} = X\sim p$.

p valid

$\Rightarrow$ Yp is valid

$\Rightarrow$ v_o is inconsistent in $T(X\sim p)$ by theorem 4.7,

$\Rightarrow$ $u_1, \ldots, u_n$ are inconsistent by consistency rule C4,

$\Rightarrow$ $\sim p_{u_i}$ for all i by theorem 4.7,

$\Rightarrow$ $\sim(p_{u_i} \vee \ldots \vee p_{u_n})$ by PC,

$\Rightarrow$ $\sim p_{v_o} X$ by lemma 4.18, PC.

$\therefore$ But $p_{v_o}X$ is just $\neg p$, so

$\vdash p$ by PC. ∎

Gabbay et al. define a proof system DUX for the logic of until on infinite paths. Their axioms are related to A8-A14, but are different due to their slightly different definitions of G and X, and the fact that their paths must be infinite. Our system was developed independently of theirs, and our completeness proof is quite different from theirs. As it is possible to express in MPL that a path is infinite, our method encompasses theirs.

As a final corollary to the decision method for MPL, we note that the LL-processes, those defined by finite LL-graphs, are complete for MPL.

Theorem 4.23. Every satisfiable MPL formula is satisfied by a model whose process is an LL-process. ∎

Chapter 5

Programs in Process Logic

In this chapter we define an extension MPL/P of MPL by adding programs to the syntax of formulas. Though MPL/P is a natural extension of MPL, MPL/P proves much more difficult to analyze than MPL. We have few results concerning MPL/P.

The main purpose of this chapter is to give a formal definition of MPL/P, an important extension of MPL, and to relate the expressive power of MPL/P to that of other logics. In judging the relative power of two logics of processes, it is only fair that either both have programs, or neither has programs. We show that MPL/P is more expressive than PDL or SOAPL, and is at least as expressive as PDL^+ and Nishimura's process logic, NL. We conjecture that MPL/P is strictly more expressive than all four of the above logics.

5.1. Definitions

In this section we define MPL/P. Programs were defined in Chapter 1. For MPL/P, we need to extend programs to labeled programs.

Labels

The usual method of reasoning about a program is to

reason about each part separately, combining the separate results to obtain a result applying to the whole. While that method works well for sequential programs, we encounter difficulties when trying to use it for concurrent programs. The behavior of α running in isolation can be so different from its behavior when running concurrently with β, that we can never divorce α from β when we reason about α//β. Nevertheless, we would like to be able to discuss α's contribution to the system α//β. We do that by giving α a name, say ℓ. By referencing ℓ, we can make statements such as:

1) In α//β, whenever α halts, p holds;

2) (finite delay) *on every infinite* path, α makes infinitely many transitions;

3) α preserves the truth of p (though β may not).

This is the sort of non-interference property which is implicit in Owicki's proof technique, but which cannot be expressed in her logic.

Labels have many different uses. It is clear that we need some means of referring to parts of a program. But it is not the purpose of this work to study the relationships between various forms of label references. Rather, we simply demonstrate what can be said with certain types of label references. Thus we feel justified in providing MPL/P with a variety of means of referring to labels. It may turn out that some are expressible

in terms of the others.

Labels have basically two different uses; as position labels, telling the current value of a program counter, and as transition labels, telling which part of a program makes a particular transition. Statement (1) above uses a position label to tell when α has terminated; that is, when α is at its final label. Statement (2) uses a transition label to determine whether α makes any transitions.

Labels are added to processes as follows. To every transition is added two sets of labels from a label set Γ. The first set consists of position labels, the second set of transition labels.

$$\Psi_\ell(U) = (P(\Gamma) \times U \times P(\Gamma) \times U)^{*+\omega},$$

$$\Pi_\ell(U) = P(\Psi_\ell(U)).$$

The operator ":" is the labeling operator. If ℓ is a label and α is a program, then $\ell:\alpha$ is a program. Every transition made by α is labeled ℓ. In $\ell_1:\ell_2:\alpha$, every transition is labeled both ℓ_1 and ℓ_2. The function π: programs $\to \Pi_\ell(U)$ is defined as follows:

1. Basic programs have the usual semantics, with $S = T = \emptyset$ in every transition $\langle S,u,T,v\rangle$. The constraints listed on page 12 apply to basic programs.

2. If p is a formula, then $\Box p?$ is a program. The box forces testable formulas to depend only on a stage and a process, not on a path. If p is already independent

of a path, then $\Box p \equiv p$, and the $\Box$ may be ignored. As are basic programs, tests are labeled $\emptyset$.

3. $\pi(\ell:\alpha)$ is obtained from $\pi(\alpha)$ by replacing every transition $\langle S,u,T,v\rangle$ by $\langle S \cup \{\ell\}, u, T \cup \{\ell\}, v\rangle$.

4. $\alpha \cup \beta$, $\alpha;\beta$ and α^* have the usual meanings.

5. The shuffle operator // must be defined so as to maintain position labels. Transition labels need no special treatment. If σ and τ are two transition sequences, define $\sigma^{(\tau)}$ to be σ, with every transition $\langle S,u,T,v\rangle$ replaced by $\langle S \cup S', u, T, v\rangle$, where $\tau = \langle S',u',T',v'\rangle\tau'$. If $\sigma_1\sigma_2 \ldots \in \pi(\alpha)$ and $\tau_1\tau_2 \ldots \in \pi(\beta)$, (any of the σ_i and τ_i are permitted to be either empty, finite or infinite) then $\sigma_1{}^{(\tau_1)}\tau_1{}^{(\sigma_2)}\sigma_2{}^{(\tau_2)}\tau_2{}^{(\sigma_3)} \ldots$ is in $\pi(\alpha//\beta)$.

The dot operator

An MPL formula describes a property of a process. Until now, we have only tested the truth of a formula with respect to the process π provided by an MPL structure. A natural extension of MPL is to let a structure provide many different processes, and to add to MPL a means of specifying which process or combination of processes is supposed to satisfy a given formula. The dot operator serves that purpose, the formula $\alpha \cdot p$ meaning "p holds for process $\pi(\alpha)$."

5.2. Formal semantics of MPL/P

An MPL/P structure is a six-tuple $A = (U, \Sigma_0, \pi_0, \Phi_0, \phi_0, \Gamma)$; where U, Φ_0 and ϕ_0 are the same as in an MPL structure, and

Σ_0 is a set of basic programs,

π_0: $\Sigma_0 \rightarrow \Pi_\ell(U)$ assigns a process to each basic program, and

Γ is a set of labels.

An MPL/P environment, providing all of the information needed to determine the truth value of any MPL/P formula, consists of a structure A, a process π, a path $\psi \in \pi$ and a stage $\tau \leq \psi$.

Let α be a labeled program, $\ell \in \Gamma$, p,q be MPL/P formulas, and $P \in \Phi_0$ be a basic formula.

1. P is an MPL/P formula. $\pi,\psi,\tau \models P$ iff $end(\tau) \in \phi_0(P)$.

2. $\sim p$, $p \vee q$ are MPL/P formulas, with the usual semantics.

3. Yp is an MPL/P formula. $\pi,\psi,\tau \models Yp$ iff $((\psi = \tau\langle S,u,T,v\rangle\psi' \text{ and } \tau\langle S,u,T,v\rangle \text{ is legal}) \Rightarrow \pi,\psi,\tau\langle S,u,T,v\rangle \models p)$.

4. pWq is an MPL/P formula. $\pi,\psi,\tau \models pWq$ iff $(\forall \tau')(\tau \leq \tau' \leq \psi \Rightarrow ((\forall \tau'')(\tau \leq \tau'' \leq \tau' \Rightarrow \pi,\psi,\tau'' \models q) \Rightarrow \pi,\psi,\tau' \models p))$.

5. $\Box p$ is an MPL/P formula. $\pi,\psi,\tau \models \Box p$ iff $(\forall \psi' \in \pi)(\psi' \geq \tau \Rightarrow \pi,\psi',\tau \models p)$.

6. $\alpha \cdot p$ is an MPL/P formula. Let $u = end(\tau)$. $\pi,\psi,\tau \models \alpha \cdot p$ iff $(\forall \psi' \in \pi(\alpha))(\psi' \geq (u,\lambda) \Rightarrow \pi(\alpha),\psi', (u,\lambda) \models p)$.

We provide a variety of formulas for referencing labels.

7. $N\ell$ means "the next transition is made by program ℓ." $\pi,\psi,\tau \models N\ell$ iff $\psi = \tau\langle S,u,T,v\rangle\psi'$ and $\ell \in T$.

8. $in(\ell)$ means "some program is executing in $\ell:\alpha$." $\pi,\psi,\tau \models in(\ell)$ iff $\psi = \tau\langle S,u,T,v\rangle\psi'$ and $\ell \in S$.

9. $@\ell$ means "some program is just ready to start $\ell:\alpha$." $\pi,\psi,\tau \models @\ell$ iff $(\tau=\tau'\langle S,u,T,v\rangle$ and $\ell \notin S$, or $\tau=(u,\lambda))$ and $(\psi=\tau\langle S',u',T',v'\rangle\psi'$ and $\ell \in S')$.

10. $end(\ell)$ means "some program has just finished $\ell:\alpha$." $\pi,\psi,\tau \models end(\ell)$ iff $\tau=\tau'\langle S,u,T,v\rangle$ and $\ell \in S$ and $(\psi = \tau\langle S',u',T',v'\rangle\psi'$ and $\ell \notin S'$, or $\psi=\tau)$.

Examples of formulas using labels are

1) whenever α terminates, p holds =

$(\ell:\alpha)//\beta \cdot \Box G(end(\ell) \supset p)$;

2) α preserves p =

$(\ell:\alpha)//\beta \cdot \Box G(p \wedge N\ell \supset Yp)$.

5.3. Expressive power of MPL/P

We begin by relating MPL/P to PDL$^+$, SOAPL, and NL. Each has been claimed (see [HP78], [Pa78], [N79]) to be a powerful logic, particularly NL, which Nishimura shows is expressively complete for a class of logics related to and including Pratt's process logic.

In each simulation, we assume that the logic being simulated is defined in an appropriate manner over MPL/P models, so that it makes sense to relate expressive powers of the logics.

PDL$^+$

The [ ] and [ ]$^+$ operators of PDL$^+$ (see [HP78]) are defined in MPL/P as follows:

$[\alpha]p \equiv \alpha \cdot \Box G(\Diamond H \supset p)$.

$[\alpha]^+p \equiv [\alpha]p \wedge \alpha \cdot \Box FY\text{false}$.

Hence MPL/P is at least as expressive as PDL$^+$. That MPL/P is more expressive than PDL follows from the fact that SOAPL is at least as expressive as PDL, and MPL/P is more expressive than SOAPL.

SOAPL

Nishimura [N79] shows that NL can simulate SOAPL, so we only need to simulate NL. That MPL/P is more expressive than SOAPL follows from Parikh's result [Pa78] that every satisfiable SOAPL formula is satisfied by a closed process. $A \cdot (FY\text{false} \wedge G \Diamond X\text{true})$, stating

that A contains no infinite paths, but A can always make more progress, is satisfied by some non-closed A, but not by any closed A.

NL

Nishimura's operator $[\alpha]$ is just our dot operator $\alpha\cdot$. Besides $[\alpha]$, NL has only until and Boolean functions, which are easily handled by MPL/P. NL has no analog to our $\Box$ operator, which leads us to conjecture that NL is weaker than MPL/P.

Recently Harel, Kozen and Parikh [HKP80] have defined a process logic PL which merges temporal logic and PDL in a way somewhat different from MPL. PL was unknown to us when we developed MPL. In PL, all formulas, including basic formulas, depend for their truth values on paths. The semantics of the PDL operator $\langle\alpha\rangle p$ is changed as follows:

$$\psi \models \langle\alpha\rangle p \quad \text{iff} \quad \exists \psi' \in \pi(\alpha)(\psi\cdot\psi' \models p).$$

Additionally, PL includes the until operator, and an operator f which is defined by

$$\psi \models fp \quad \text{iff} \quad \text{start}(\psi) \models p.$$

The relation between PL and MPL/P is not at all clear. Due to our result that nonstandard and standard semantics produce the same satisfiable formulas, it might not be important that basic formulas are interpreted over paths

in PL. But MPL/P does not appear to be able to simulate $\langle\alpha\rangle p$, due to the fact that $\langle\alpha\rangle p$ depends on an entire path, not just its final state. Conversely, PL does not appear to have any means of expressing branching time properties of programs.

Finite delay

Our // operator permits one component to run forever, to the exclusion of the other. In some applications we may want to assume that // is fair, so that any component which is active eventually gets to run a step. Even with our unfair // operator, MPL/P can be used to discuss programs with a fair // operator. For example, suppose our fairness criterion is that, on every infinite path in $\alpha//\beta$, both α and β make infinitely many transitions. Let

$$FD(\ell) \equiv (GXtrue \supset GFN\ell).$$

Then

$$(\ell_1:\alpha)//(\ell_2:\beta)\cdot(FD(\ell_1) \wedge FD(\ell_2) \supset p)$$

states that every fair path in $\alpha//\beta$ obeys p. $FD(\ell)$ is an over simple fairness criterion. A more reasonable one takes into account that one of the components may terminate or remain blocked forever. A statement which takes into account those possibilities is

$$FD_2(\ell) \equiv GXtrue \supset (Fend(\ell) \vee GFN\ell \vee FG\Box\sim N\ell),$$

which states that, on every infinite path, either program ℓ terminates, or it makes infinitely many transitions, or beyond some stage it is never possible for ℓ to make the next transition, even on a different path.

Partial correctness proofs

One test of the power of a logic is whether existing proofs can be carried out within that logic. In order to use a particular proof method, not only the end results but all of the intermediate results must be expressible in the logic. Suitable proof rules can then be written.

Owicki [OG76] gives a proof system for proving partial correctness assertions about concurrent programs. A very important notion in her proof system is that of non-interference; that is, in $\alpha//\beta$, no step of α can cause p to change from true to false. We have shown above that non-interference can be expressed in MPL/P. Owicki's logic provides no mechanism for expressing non-interference, with the result that non-interference must be added artificially to a proof rule, whose antecedents are not formulas, but are proofs. By expressing non-interference in MPL/P, we carry out simulations of Owicki-style proofs, using the usual sort of proof rules, which prove certain formulas, given certain other formulas. Further-

more, we are permitted greater flexibility. If we have designed our program so that β does not interfere with α because β preserves p, we can prove, once and for all, that β preserves p in α//β. We might then show that α//A works correctly (for some suitable meaning of "correct") whenever A preserves p.

5.4. Conclusion

We have shown that MPL/P is a powerful logic of processes. Moreover, with such statements as "α cannot deadlock," written in MPL/P as $\alpha \cdot \Box G(H \vee \Diamond X\text{true})$, and the statement that all finite delay paths of α//β obey p, we have shown that at least a good part of the power of MPL/P is needed. Any logic of processes which is less expressive than MPL/P should have its lack of power justified, whether to permit analysis, or because for a certain application the full power of MPL/P is not needed.

We have no decision method or proof system for MPL/P. The tableau method used for MPL does not readily extend to MPL/P. It seems unlikely that the addition of programs to MPL results in an undecidable logic. The existence of a complete proof system for PL [HKP80], which appears to have some features in common with MPL/P, is encouraging.

REFERENCES

A79 Abrahamson, K., "Modal logic of concurrent nondeterministic programs," Lecture Notes in Computer Science 70, ed. G. Kahn, Springer-Verlag, Berlin, Heidelberg, New York (1979), 21-33.

A80 Abrahamson, K., "Boolean variables in regular expressions and finite automata," Tech. Report 80-08-02, Univ. of Wash., Seattle, Washington, 1980.

AHU74 Aho, V. A., J. E. Hopcroft and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass.

BP78 Berman, F. and G. L. Peterson, "Expressiveness hierarchy for PDL with rich tests (extended abstract)," manuscript, Department of Computer Science, University of Washington, Seattle (1978).

CS76 Chandra, A. K. and L. J. Stockmeyer, "Alternation," Proceedings of 17th IEEE Symposium on Foundations of Computer Science (1976), 98-108.

Co70 Cohen, R. S., "Star height of certain families of regular events," JCSS 4 (1970), 281-297.

D78 Dijkstra, E. W., L. Lamport, A. J. Martin, C. S. Scholten and E. M. F. Steffens, "On-the-fly garbage collection: an exercise in Cooperation," CACM 21, 11 (1978), 966-975.

F167 Floyd, R. N., "Assigning meanings to programs," Proceedings of Symposium on Applied Math. 19, (1967), 19-32.

FL79 Fischer, M. J. and R. E. Ladner, "Propositional dynamic logic of regular programs," JCSS 18 (1979), 194-211.

FP76 Francez, N. and A. Pnueli, "A proof method for cyclic programs," Proceedings of IEEE International Conference on Parallel Processing (1976), 235-245.

GPSS80 Gabbay, D., A. Pnueli, S. Shelah and J. Stavi, "On the temporal analysis of fairness," Proceedings of 7th Symposium on Principles of Programming Languages (1980), 163-173.

GS65 Ginsburg, S. and E. H. Spanier, "Mappings of languages by two tape devices," J. ACM 12 (1965), 423-434.

HKP80 Harel, D., D. Kozen and R. Parikh, "Process logic: expressiveness, decidability, completeness (extended abstract)," manuscript, IBM Watson Research Center, Yorktown Heights, New York (1980).

HP78 Harel, D. and V. R. Pratt, "Nondeterminism in logics of programs," Proceedings of 5th Annual ACM Symposium on Principles of Programming Languages (1978), 203-213.

Ho69 Hoare, C. A. R., "An axiomatic basis for computer programming," CACM 12, (1969), 576-580.

Ho76 Hoare, C. A. R., "Communicating sequential processes," CACM 21,8 (1978) 666-677.

HC68 Hughes, G. E. and M. J. Cresswell, An Introduction to Modal Logic, London, Methuen, 1968.

K68 Kamp, J. A. W., "Tense logic and the theory of linear order," Univ. of California, Los Angeles, Ph.D. thesis, 1968.

Ko76 Kozen, D., "On parallelism in Turing machines." Proceedings of 17th IEEE Symposium on Foundations of Computer Science (1976), 89-97.

L80 Lamport, L., "'Sometime' is sometimes 'not never'," Proceedings of 7th Annual ACM Symposium on Principles of Programming Languages (1980), 174-185.

LF79 Lynch, N. A. and M. J. Fischer, "On describing the behavior and implementation of distributed systems," Lecture Notes in Computer Science 70, ed. G. Kahn, Springer-Verlag, Berlin, Heidelberg, New York (1979), 147-171.

MW78 Manna, Z. and R. Waldinger, "Is 'sometime' sometimes better than 'always'?," CACM 21,2 (1978), 159-172.

M74 Meyer, A. R., "Weak monadic second order theory of successor is not elementary recursive," Lecture Notes in Mathematics 453, Springer-Verlag (1975), 132-154.

MM77 Milne, G. and R. Milner, "Concurrent processes and their syntax," Internal Report CSR-2-77, Department of Computer Science, Edinburg, May 1977.

N79 Nishimura, H., "Descriptively complete process logic," manuscript, Kyoto University, Kyoto, Japan (1979).

OG76 Owicki, S. and D. Gries, "An axiomatic proof technique for parallel programs," Acta Informatica 6,4 (1976), 319-340.

Ow78 Owicki, S., Colloquium presentation, Univ. of Washington (1978).

Pa78 Parikh, R., "A decidability result for second order process logic," Proceedings of 19th IEEE Symposium on Foundations of Computer Science (1978), 177-183.

Pn77 Pnueli, A., "The temporal logic of programs," Proceedings of 18th IEEE Symposium on Foundations of Computer Science (1977), 46-57.

Pn79 Pnueli, A., "The temporal semantics of concurrent programs," Lecture Notes in Computer Science 70, ed. G. Kahn, Springer-Verlag, Berlin, Heidelberg, New York (1979), 1-20.

Pr76 Pratt, V. R., "Semantical considerations on Floyd-Hoare logic," Proceedings of 17th IEEE Symposium on Foundations of Computer Science (1976), 109-121.

Pr78 Pratt, V. R., "A practical decision method for propositional dynamic logic," 10th Annual ACM Symposium on Theory of Computing (1978), 326-337.

Pr80 Pratt, V. R., "Flowgraph logic and the elimination of Kleene elimination," manuscript, Department of Computer Science, Massachusetts Institute of Technology, April, 1980.

R69 Rabin, M. O., "Decidability of second order theories and automata on infinite trees," Transactions of American Mathematical Society 141 (1969), 1-35.

RP80 Reif, J. H. and G. L. Peterson, "A dynamic logic of multiprocessing with incomplete information," Proceedings of 7th Annual ACM Symposium on Principles of Programming Languages (1980), 193-202.

St74 Stockmeyer, L. J., "The complexity of decision problems in automata theory and logic," Ph.D. thesis, Massachusetts Institute of Technology, 1974.

Vita

Karl Raymond Abrahamson

Born November 5, 1953 in Palo Alto, California

Education:

Cubberly Senior High School, Palo Alto, California

9/69 - 6/71

Foothill Community College, Los Altos Hills, California

9/71-6/73

University of Washington, Seattle, Washington

9/73-6/75

University of Washington, Seattle, Washington

9/75-9/80

Degrees:

Bachelor of Science from University of Washington (Department of Mathematics), 9/75

DISTRIBUTION LIST

Office of Naval Research Contract N00014-80-C-0221
Michael J. Fischer, Principal Investigator

Defense Documentation Center
Cameron Station
Alexandria, VA 22314
(12 copies)

Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Dr. R. B. Grafton, Scientific Officer (1 copy)
Information Systems Program (437) (2 copies)
Code 200 (1 copy)
Code 455 (1 copy)
Code 458 (1 copy)

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106
(1 copy)

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D.C. 20375
(6 copies)

Office of Naval Research
Resident Representative
University of Washington, JD-27
422 University District Building
1107 NE 45th Street
(1 copy)

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
Code RD-1
Washington, D.C. 20380
(1 copy)

Naval Ocean Systems Center
Advanced Software Technology Division
Code 5200
San Diego, CA 92152
(1 copy)

Mr. E. H. Gleissner
Naval Ship Research and Development Center
Computation and Mathematics Dept.
Bethesda, MD 20084
(1 copy)

Captain Grace M. Hooper (008)
Naval Data Automation Command
Washington Navy Yard
Building 166
Washington, D.C. 20374
(1 copy)

Defense Advanced Research Projects Agency
Attn: Program Management/MIS
1400 Wilson Boulevard
Arlington, VA 22209
(3 copies)


END
DATE
FILMED